



---

# Cadence® NC-Verilog® Simulator Help

## Product Version 3.4 January 2002

© 1995-2001 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.

**cadence**

---

# **Contents**

---

## **1**

<b><u>Overview of the Cadence® NC-Verilog® Simulator</u></b>	25
<u>Native Compiled Code</u>	25
<u>The Interleaved Native Compiled Code (INCA) Architecture</u>	26
<u>The Cadence Desktop Simulators</u>	28
<u>Language Support</u>	29
<u>Memory Requirements</u>	30
<u>Setting Up Your Design Environment</u>	30
<u>Running the NC-Verilog Simulator</u>	31
<u>Single-Step Invocation With ncverilog</u>	37
<u>Multi-Step Invocation (Library-Based Mode)</u>	38
<u>Simulator Library Databases</u>	38

## **2**

<b><u>Getting Help</u></b>	41
<u>About Online Help</u>	41
<u>Invoking the Documentation System</u>	42
<u>Printing Documents</u>	43
<u>Searching Documents</u>	43
<u>Getting Help on Commands to Run Tools</u>	43
<u>Getting Help on Simulator Commands</u>	44
<u>Getting Help on Tool Messages</u>	44
<u>Related Manuals and Specifications and Other Documentation</u>	45
<u>Verilog</u>	45
<u>VHDL</u>	46
<u>Other Documentation</u>	46

## 3

<u>Using NCLaunch</u> .....	48
-----------------------------	----

## 4

<u>Running NC-Verilog with the ncverilog Command</u> .....	50
--	----

<u>Overview</u> .....	51
-----------------------	----

<u>How ncverilog Works</u> .....	53
----------------------------------	----

<u>ncverilog Command Syntax</u> .....	55
---------------------------------------	----

<u>ncverilog Command Options</u> .....	56
--	----

<u>Plus Options for NC-Verilog Tools</u> .....	66
--	----

<u>Verilog-XL Command-Line Options Translation</u> .....	70
--	----

<u>Verilog-XL Dash (-) Option Translation Table</u> .....	70
---	----

<u>Verilog-XL Plus (+) Option Translation Table</u> .....	72
---	----

<u>Example ncverilog Run</u> .....	75
------------------------------------	----

<u>Mapping of Warning Messages</u> .....	77
--	----

<u>Updating Design Changes</u> .....	78
--------------------------------------	----

<u>Using +ncuid+ to Run in Regression Mode</u> .....	79
--	----

<u>Using -R and -r to Simulate a Snapshot</u> .....	81
---	----

<u>Using -R to Simulate a Snapshot Multiple Times</u> .....	81
---	----

<u>Using -r to Simulate a Saved Snapshot</u> .....	83
--	----

<u>PLI Tasks</u> .....	85
------------------------	----

<u>SDF Annotation</u> .....	85
-----------------------------	----

<u>Using \$test\$plusargs to Selectively Perform Annotations</u> .....	87
--	----

<u>Specifying Precision</u> .....	87
-----------------------------------	----

<u>Turning Off SDF Annotation</u> .....	87
---	----

<u>Using the +sdf cmd file+ Option</u> .....	88
--	----

<u>Controlling SDF Annotator Output</u> .....	88
---	----

<u>Troubleshooting</u> .....	89
------------------------------	----

## 5

<u>Modeling Your Hardware</u> .....	90
-------------------------------------	----

<u>Arrays of Instances</u> .....	91
----------------------------------	----

<u>Instance Array Names</u> .....	91
-----------------------------------	----

<u>Array Range Expressions</u> .....	91
--------------------------------------	----

## NC-Verilog Simulator Help

---

<u>Port Connections</u>	92
<u>Ports of Instance Arrays</u>	93
<u>Differing Instances in an Array</u>	94
<u>Hierarchical References</u>	94
<u>NC-Verilog Restrictions to the IEEE Standard</u>	95
<u>Arrays of Instances and Tcl Commands</u>	98
<u>Verilog IEEE Std 1364-2001 Enhancements</u>	99
<u>Comma-Separated Sensitivity List</u>	99
<u>File I/O Enhancements</u>	99

## **6**

<b><u>Setting Up Your Environment</u></b>	107
<u>Overview</u>	108
<u>The Library.Cell:View Approach</u>	109
<u>The cds.lib File</u>	110
<u>The Work Library</u>	111
<u>cds.lib Statements</u>	111
<u>cds.lib Syntax Rules</u>	113
<u>Example cds.lib File</u>	114
<u>Binding One Library to Multiple Directories</u>	114
<u>Debugging cds.lib Files</u>	116
<u>The hdl.var File</u>	118
<u>hdl.var Statements</u>	120
<u>hdl.var Variables</u>	121
<u>hdl.var Syntax Rules</u>	127
<u>Example hdl.var File</u>	129
<u>Debugging hdl.var Files</u>	129
<u>The setup.loc File</u>	131
<u>setup.loc Syntax Rules</u>	132
<u>Directory Structure Example</u>	133

## **7**

<b><u>Compiling Verilog Source Files with ncvlog</u></b>	137
<u>Overview</u>	138
<u>ncvlog Command Syntax</u>	140

## NC-Verilog Simulator Help

---

<u>ncvlog Command Options</u>	141
<u>Example ncvlog Command Lines</u>	158
<u>hdl.var Variables</u>	160
<u>Conditionally Compiling Source Code</u>	163
<u>Controlling the Compilation of Design Units into Library.Cell:View</u>	165
<u>The Default</u>	165
<u>The LIB_MAP and VIEW_MAP Variables</u>	167
<u>The WORK and VIEW Variables</u>	169
<u>The -work and -view Options</u>	171
<u>The -specificunit Option</u>	172
<u>The `worklib and `view Compiler Directives</u>	173
<u>Mapping of Modules Defined Within `include Files</u>	175
<u>Defining Macros on the Command Line</u>	178

## 8

<u>Elaborating the Design with ncelab</u>	180
<u>Overview</u>	181
<u>ncelab Command Syntax</u>	185
<u>General Options</u>	185
<u>VHDL Only Options</u>	186
<u>Verilog Only Options</u>	186
<u>ncelab Command Options</u>	188
<u>Example ncelab Command Lines</u>	227
<u>hdl.var Variables</u>	232
<u>How Modules and UDPs Are Resolved During Elaboration</u>	233
<u>The Default Binding Mechanism</u>	234
<u>The -binding Option</u>	239
<u>The 'uselib Compiler Directive</u>	242
<u>Enabling Read, Write, or Connectivity Access to Simulation Objects</u>	248
<u>Regression Mode and Tcl Commands</u>	249
<u>Regression Mode and PLI/VPI/VHPI Applications</u>	249
<u>Regression Mode and the SimVision Debug Environment</u>	251
<u>Using -access to Specify Read/Write/Connectivity Access</u>	251
<u>Using -afile to Include an Access File</u>	253
<u>Generating an Access File</u>	260

## NC-Verilog Simulator Help

---

<u>Guidelines for Access Control</u>	261
<u>Disabling Timing in Selected Portions of a Design</u>	267
<u>Selecting a Delay Mode</u>	270
<u>Delay Modes</u>	270
<u>Reasons to Select a Delay Mode</u>	272
<u>Setting a Delay Mode</u>	272
<u>Timescales and Simulation Time Units</u>	273
<u>Overriding Delay Values</u>	275
<u>Delay Mode Example</u>	277
<u>Decompiling with Delay Modes</u>	278
<u>Macro Module Expansion and Delay Modes</u>	278
<u>Summary of Delay Mode Rules</u>	279
<u>Setting Pulse Controls</u>	280
<u>Overview</u>	280
<u>Global Pulse Control</u>	281
<u>Pulse Control for Specific Modules and Module Paths</u>	283
<u>Pulse Filtering Style</u>	285

## **9**

<u>Simulating Your Design with ncsim</u>	296
<u>Overview</u>	297
<u>ncsim Command Syntax</u>	299
<u>ncsim Command Options</u>	301
<u>Example ncsim Command Lines</u>	320
<u>hdl.var Variables</u>	321
<u>Invoking the Simulator</u>	322
<u>Invoking the Simulator in Noninteractive Mode</u>	323
<u>Invoking the Simulator in Interactive Mode</u>	324
<u>Starting a Simulation</u>	325
<u>Saving, Restarting, Resetting, and Reinvoking a Simulation</u>	327
<u>Saving and Restarting the Simulation</u>	327
<u>Resetting the Simulation</u>	329
<u>Reinvoking a Simulation</u>	330

## NC-Verilog Simulator Help

---

<u>Updating Design Changes When You Invoke the Simulator</u>	332
<u>Providing Interactive Commands from a File</u>	335
<u>-input Command Syntax</u>	336
<u>Exiting the Simulation</u>	338
 <b>10</b>	
<u>Mixed Verilog/VHDL Simulation</u>	339
<u>Mapping of Data Types</u>	340
<u>Importing Verilog into VHDL</u>	351
<u>Using Default Binding</u>	353
<u>Using a Configuration Specification or Configuration Declaration</u>	358
<u>Using Direct Instantiation</u>	366
<u>Using a Shell</u>	369
<u>Importing VHDL into Verilog</u>	373
<u>Importing VHDL into Verilog without a Shell</u>	375
<u>Importing VHDL into Verilog with a Shell</u>	377
<u>Importing VHDL into Verilog with ncverilog</u>	379
<u>A Verilog-VHDL-Verilog Example</u>	380
<u>Preparing the Design without Shells</u>	380
<u>Preparing the Design with Shells</u>	383
<u>Generating a Shell with ncshell</u>	385
<u>ncshell Command Syntax</u>	385
<u>ncshell Command Options</u>	387
<u>Importing a Verilog-XL Design into VHDL</u>	393
<u>Preparing a Leapfrog Verilog Model Import Design</u>	399
<u>The ncximport Utility</u>	399
<u>ncximport Syntax</u>	400
<u>ncximport Command Options</u>	400
<u>Mixed-Language Out-of-Module References</u>	405
<u>Path Names and Mixed-Language Designs</u>	408
<u>SDF Annotation for Mixed-Language Designs</u>	410
<u>Generating a Value Change Dump (VCD) File for a Mixed-Language Design</u>	410

### **11**

<b><u>Debugging Your Design</u></b>	416
<b><u>Managing Databases</u></b>	418
<b><u>Opening a Database</u></b>	418
<b><u>Displaying Information About Databases</u></b>	419
<b><u>Disabling a Database</u></b>	419
<b><u>Enabling a Database</u></b>	419
<b><u>Closing a Database</u></b>	420
<b><u>Setting and Deleting Probes</u></b>	421
<b><u>Setting a Probe</u></b>	421
<b><u>Displaying Information About Probes</u></b>	422
<b><u>Disabling a Probe</u></b>	422
<b><u>Enabling a Probe</u></b>	423
<b><u>Deleting a Probe</u></b>	423
<b><u>Traversing the Model Hierarchy</u></b>	424
<b><u>Path Names</u></b>	424
<b><u>Setting the Debug Scope</u></b>	424
<b><u>Setting Breakpoints</u></b>	426
<b><u>Setting a Condition Breakpoint</u></b>	427
<b><u>Setting a Source Code Line Breakpoint</u></b>	428
<b><u>Setting an Object Breakpoint</u></b>	429
<b><u>Setting a Time Breakpoint</u></b>	430
<b><u>Setting a Delta Breakpoint</u></b>	431
<b><u>Setting a Process Breakpoint</u></b>	431
<b><u>Setting a Subprogram Breakpoint</u></b>	432
<b><u>Disabling, Enabling, Deleting, and Displaying Breakpoints</u></b>	433
<b><u>Stepping Through Lines of Code</u></b>	434
<b><u>Forcing and Releasing Signal Values</u></b>	435
<b><u>Depositing Values to Signals</u></b>	436
<b><u>Displaying Information About Simulation Objects</u></b>	437
<b><u>Displaying the Drivers of Signals</u></b>	437
<b><u>Checking for Bus Contention and Bus Float Conditions</u></b>	438
<b><u>Displaying Waveforms with SimVision Waveform Viewer</u></b>	440
<b><u>Creating an SHM Database and Probing Signals</u></b>	440
<b><u>Opening a Database with \$shm_open</u></b>	441

## NC-Verilog Simulator Help

---

<u>Probing Signals with \$shm_probe</u>	442
<u>Invoking SimVision Waveform Viewer</u>	444
<u>Using \$recordvars and Related Tasks</u>	446
<u>Generating a Value Change Dump (VCD) File</u>	458
<u>Generating a VCD File with Tcl Commands</u>	458
<u>Generating a VCD File with VCD System Tasks</u>	460
<u>Syntax and Format of the VCD File</u>	462
<u>Generating an Extended Value Change Dump (EVCD) File</u>	463
<u>Using the \$dumports System Task</u>	463
<u>Using the \$dumports_close System Task</u>	465
<u>Syntax and Format of the EVCD File</u>	466
<u>Using the format_flag Argument to Control \$dumports Output</u>	473
<u>\$dumports Restrictions</u>	476
<u>Comparing Databases with Comparescan</u>	477
<u>Generating a Code Coverage Database File</u>	478
<u>Displaying Debug Settings</u>	480
<u>Setting a Default Radix</u>	480
<u>Setting Variables</u>	481
<u>Suppressing Assert Messages in IEEE or User-Defined Packages</u>	486
<u>Editing a Source File</u>	488
<u>Searching for a Line Number in the Source Code</u>	489
<u>Searching for a Text String in the Source Code</u>	489
<u>Configuring Your Simulation Environment</u>	489
<u>Saving and Restoring Your Simulation Environment</u>	490
<u>Creating or Deleting an Alias</u>	491
<u>Getting a History of Commands</u>	491
<u>Managing Custom Buttons</u>	492
<b>12</b>	
<b><u>Using the Tcl Command-Line Interface</u></b>	493
<b><u>Overview</u></b>	493
<b><u>Executing UNIX Commands</u></b>	494
<b><u>Using Wildcards Characters in Tcl Commands</u></b>	495
<b><u>Command Description Conventions</u></b>	496

## NC-Verilog Simulator Help

---

<u>alias</u> .....	498
<u>alias Command Syntax</u> .....	498
<u>alias Command Modifiers and Options</u> .....	498
<u>alias Command Examples</u> .....	499
<u>attribute</u> .....	500
<u>attribute Command Syntax</u> .....	500
<u>attribute Command Options and Modifiers</u> .....	501
<u>attribute Command Examples</u> .....	501
<u>call</u> .....	504
<u>call Command Syntax</u> .....	504
<u>call Command Modifiers and Options</u> .....	506
<u>call Command Examples</u> .....	507
<u>check</u> .....	508
<u>check Command Syntax</u> .....	508
<u>check Command Modifiers and Options</u> .....	509
<u>check Command Examples</u> .....	512
<u>coverage</u> .....	516
<u>database</u> .....	517
<u>database Command Syntax</u> .....	519
<u>database Command Modifiers and Options</u> .....	520
<u>Opening a Database</u> .....	520
<u>Setting a Database As the Default</u> .....	524
<u>Displaying Information about Databases</u> .....	524
<u>Disabling a Database</u> .....	524
<u>Enabling a Database</u> .....	525
<u>Closing a Database</u> .....	525
<u>database Command Examples</u> .....	525
<u>deposit</u> .....	528
<u>deposit Command Syntax</u> .....	529
<u>deposit Command Modifiers and Options</u> .....	529
<u>deposit Command Examples</u> .....	530
<u>describe</u> .....	532
<u>describe Command Syntax</u> .....	532
<u>describe Command Modifiers and Options</u> .....	532
<u>describe Command Examples</u> .....	533

## NC-Verilog Simulator Help

---

<u>drivers</u>	.....	537
<u>drivers Command Syntax</u>	.....	537
<u>drivers Command Modifiers and Options</u>	.....	537
<u>drivers Command Report Format</u>	.....	538
<u>drivers Command Examples</u>	.....	542
<u>exit</u>	.....	548
<u>exit Command Syntax</u>	.....	548
<u>exit Command Modifiers and Options</u>	.....	548
<u>exit Command Examples</u>	.....	548
<u>finish</u>	.....	549
<u>finish Command Syntax</u>	.....	549
<u>finish Command Modifiers and Options</u>	.....	549
<u>finish Command Examples</u>	.....	549
<u>fmibkpt</u>	.....	550
<u>fmibkpt Command Syntax</u>	.....	550
<u>fmibkpt Command Modifiers and Options</u>	.....	550
<u>force</u>	.....	551
<u>force Command Syntax</u>	.....	552
<u>force Command Modifiers and Options</u>	.....	552
<u>force Command Examples</u>	.....	552
<u>help</u>	.....	554
<u>help Command Syntax</u>	.....	554
<u>help Command Modifiers and Options</u>	.....	555
<u>help Command Examples</u>	.....	555
<u>history</u>	.....	557
<u>history Command Syntax</u>	.....	557
<u>history Command Options</u>	.....	557
<u>history Command Examples</u>	.....	558
<u>input</u>	.....	560
<u>input Command Syntax</u>	.....	561
<u>input Command Modifiers and Options</u>	.....	561
<u>input Command Examples</u>	.....	561
<u>memory</u>	.....	563
<u>memory Command Syntax</u>	.....	565
<u>memory Command Modifiers and Options</u>	.....	565
<u>Loading VHDL Memory</u>	.....	565

## NC-Verilog Simulator Help

---

<u>Dumping VHDL Memory</u>	566
<u>memory Command Examples</u>	566
<u>omi</u>	568
<u>omi Command Syntax</u>	568
<u>omi Command Modifiers and Options</u>	569
<u>Displaying Information</u>	569
<u>Issuing Commands</u>	569
<u>omi Command Examples</u>	570
<u>probe</u>	571
<u>probe Command Syntax</u>	573
<u>probe Command Modifiers and Options</u>	574
<u>Creating a Probe</u>	574
<u>Deleting a Probe</u>	580
<u>Disabling a Probe</u>	580
<u>Enabling a Probe</u>	581
<u>Saving a Script to Recreate Probes</u>	581
<u>Displaying Information About Probes</u>	581
<u>probe Command Examples</u>	582
<u>process</u>	587
<u>process Command Syntax</u>	587
<u>process Command Modifiers and Options</u>	588
<u>process Command Examples</u>	589
<u>release</u>	592
<u>release Command Syntax</u>	592
<u>release Command Modifiers and Options</u>	593
<u>release Command Examples</u>	593
<u>reset</u>	594
<u>reset Command Syntax</u>	594
<u>reset Command Modifiers and Options</u>	594
<u>reset Command Examples</u>	594
<u>restart</u>	595
<u>restart Command Syntax</u>	596
<u>restart Command Modifiers and Options</u>	596
<u>restart Command Examples</u>	596

## NC-Verilog Simulator Help

---

<u>run</u>	.....	598
<u>run Command Syntax</u>	.....	598
<u>run Command Modifiers and Options</u>	.....	599
<u>run Command Examples</u>	.....	600
<u>save</u>	.....	602
<u>save Command Syntax</u>	.....	604
<u>save Command Modifiers and Options</u>	.....	604
<u>save Command Examples</u>	.....	604
<u>scope</u>	.....	608
<u>scope Command Syntax</u>	.....	608
<u>scope Command Modifiers and Options</u>	.....	609
<u>scope Command Examples</u>	.....	611
<u>source</u>	.....	616
<u>source Command Syntax</u>	.....	616
<u>source Command Modifiers and Options</u>	.....	616
<u>source Command Examples</u>	.....	617
<u>stack</u>	.....	618
<u>stack Command Syntax</u>	.....	618
<u>stack Command Modifiers and Options</u>	.....	618
<u>stack Command Examples</u>	.....	619
<u>status</u>	.....	623
<u>status Command Syntax</u>	.....	623
<u>status Command Modifiers and Options</u>	.....	623
<u>status Command Examples</u>	.....	623
<u>stop</u>	.....	624
<u>stop Command Syntax</u>	.....	625
<u>stop Command Modifiers and Options</u>	.....	626
<u>Creating a Breakpoint</u>	.....	626
<u>Deleting a Breakpoint</u>	.....	631
<u>Disabling a Breakpoint</u>	.....	632
<u>Enabling a Breakpoint</u>	.....	632
<u>Displaying Information About Breakpoints</u>	.....	632
<u>stop Command Examples</u>	.....	633
<u>Tcl Expressions as Arguments</u>	.....	641
<u>strobe</u>	.....	644
<u>strobe Command Syntax</u>	.....	645

## NC-Verilog Simulator Help

---

<u>strobe Command Modifiers and Options</u>	645
<u>strobe Command Examples</u>	646
<u>task</u>	651
<u>task Command Syntax</u>	651
<u>task Command Modifiers and Options</u>	652
<u>task Command Examples</u>	652
<u>time</u>	655
<u>time Command Syntax</u>	655
<u>time Command Modifiers and Options</u>	656
<u>time Command Examples</u>	656
<u>value</u>	658
<u>value Command Syntax</u>	658
<u>value Command Modifiers and Options</u>	659
<u>value Command Examples</u>	660
<u>version</u>	663
<u>version Command Syntax</u>	663
<u>version Command Modifiers and Options</u>	663
<u>version Command Examples</u>	663
<u>where</u>	664
<u>where Command Syntax</u>	664
<u>where Command Modifiers and Options</u>	664
<u>where Command Examples</u>	664
<u>Verilog-XL and NC-Verilog Simulator Interactive Debug Commands</u>	665

## 13

<b><u>Using the SimVision Analysis Environment</u></b>	671
--	-----

## 14

<b><u>Maximizing Simulation Performance</u></b>	673
<u>Coding Style Guidelines</u>	674
<u>General Guidelines</u>	674
<u>Recommended Verilog Coding Practices</u>	675
<u>Coding Styles to Avoid</u>	682
<u>Refining the Testbench Strategy</u>	689
<u>Use C and Tcl</u>	689

## NC-Verilog Simulator Help

---

<u>Use \$readmemb or \$readmemh for Vectors</u>	689
<u>Use \$test\$plusargs for Conditional Code</u>	692
<u>Create Self-Checking Tests</u>	693
<u>Avoiding Unnecessary Recompilation</u>	694
<u>Run the Parser in Update Mode (ncvlog -update)</u>	694
<u>Eliminate Cross-File Inheritance</u>	695
<u>Use One Module per File</u>	696
<u>Avoid Modules in `include Files</u>	696
<u>Avoid Compile-Time Conditional Code</u>	696
<u>Using the Profiler to Identify and Eliminate Simulation Bottlenecks</u>	697
<u>Stream Counts</u>	698
<u>Most Active Modules</u>	701
<u>Stream Type Summary Counts</u>	702

## **15**

<b><u>Timing Checks</u></b>	703
<u>Overview</u>	704
<u>Timing Check System Tasks</u>	705
<u>\$setup</u>	705
<u>\$hold</u>	706
<u>\$setuphold</u>	707
<u>\$width</u>	711
<u>\$period</u>	713
<u>\$skew</u>	715
<u>\$recovery</u>	716
<u>\$removal</u>	717
<u>\$recrrem</u>	719
<u>\$nochange</u>	722
<u>Using Edge-Control Specifiers</u>	722
<u>Using Notifiers</u>	723
<u>Enabling Timing Checks with Conditioned Events</u>	726
<u>Negative Timing Check Limits in \$setuphold and \$recrrem</u>	728
<u>Effects of Delayed Signals on Timing Checks</u>	729
<u>Calculation of Delayed Signals and Limit Modification</u>	731
<u>Non-Convergence in Timing Checks</u>	734

## NC-Verilog Simulator Help

---

<u>Explicitly Defining Delayed Signals</u>	739
<u>Effects of Delayed Signals on Path Delays</u>	740
<u>Restrictions</u>	742
<u>Exception Handling</u>	743
<u>Timing Violation Messages</u>	743
<u>SDF Annotation of Timing Checks</u>	745
<u>Referencing Verilog HDL Source Constructs</u>	745
<u>\$setuphold Timing Checks</u>	746
 <b>16</b>	
<u>Interconnect and Module Path Delays</u>	748
<u>Interconnect Delays</u>	748
<u>Default Interconnect Delays</u>	749
<u>Interconnect Delays and -intermod_path</u>	754
<u>Pulse Handling</u>	754
<u>SDF Annotation of Interconnect Delays</u>	755
<u>PLI Annotation of Interconnect Delays</u>	755
<u>Module Path Delays</u>	755
<u>Specify Blocks</u>	758
<u>Describing Module Paths</u>	759
<u>Establishing Full or Parallel Connection Paths</u>	768
<u>Assigning Delays to Module Paths</u>	771
<u>Selecting a Delay When Multiple Delays Are Specified for a Path</u>	775
<u>Specify Properties for Module Path Delays</u>	776
<u>Mixing Module Path Delays and Distributed Delays</u>	784
<u>Strength Changes on Path Inputs</u>	785
<u>Driving Wired Logic Outputs</u>	785
<u>Simulating Path Outputs That Drive Other Path Outputs</u>	786
<u>SDF Annotation of Module Path Delays</u>	787
 <b>17</b>	
<u>SDF Timing Annotation</u>	789
<u>VITAL SDF Annotation</u>	790
<u>Compiling the SDF File</u>	790
<u>Writing an SDF Command File</u>	791

## NC-Verilog Simulator Help

---

<u>Specifying an SDF Command File</u>	795
<u>Controlling SDF Annotator Output</u>	795
<u>Multi-Source Interconnect Delays During VITAL SDF Annotation</u>	795
<u>Command-Line Options that Affect SDF Annotation</u>	798
<b>Verilog SDF Annotation</b>	801
<u>Overview of Verilog SDF Annotation</u>	801
<u>\$sdf annotate System Task</u>	803
<u>\$sdf annotate Examples</u>	805
<u>Requirements for \$sdf annotate System Tasks</u>	811
<u>Using a Configuration File</u>	813
<u>Using an SDF Command File</u>	824
<u>Controlling SDF Annotator Output</u>	826
<u>Command-Line Options that Affect SDF Annotation</u>	826
<u>SDF Annotation for Mixed-Language Designs</u>	830

## **18**

<b>Utilities</b>	831
<u>ncexport</u>	833
<u>ncexport Command Syntax</u>	834
<u>ncexport Command Options</u>	835
<u>Example ncexport Command Lines</u>	838
<u>Example</u>	838
<u>nchelp</u>	842
<u>nchelp Command Syntax</u>	843
<u>nchelp Command Options</u>	843
<u>Example nchelp Command Lines</u>	845
<u>ncls</u>	847
<u>Listing Objects</u>	847
<u>ncls Command Syntax</u>	851
<u>ncls Command Options</u>	853
<u>Example ncls Command Lines</u>	860
<u>ncpack</u>	861
<u>ncpack Command Syntax</u>	861
<u>ncpack Command Options</u>	862
<u>Example ncpack Command Lines</u>	866

## NC-Verilog Simulator Help

---

<u>ncprep</u> .....	867
<u>ncprep Command Syntax</u> .....	868
<u>ncprep Command Options</u> .....	869
<u>Example nprep Run</u> .....	872
<u>Example nprep Output Files</u> .....	874
<u>Verilog-XL Command-Line Options Translation</u> .....	878
<u>Using Interactive Debugging Commands</u> .....	881
<u>PLI Tasks</u> .....	883
<u>SDF Annotation</u> .....	883
<u>Troubleshooting</u> .....	887
<u>ncrm</u> .....	901
<u>ncrm Command Syntax</u> .....	901
<u>ncrm Command Options</u> .....	902
<u>Example ncrm Command Lines</u> .....	904
<u>ncsdfc</u> .....	906
<u>ncsdfc Command Syntax</u> .....	907
<u>ncsdfc Command Options</u> .....	908
<u>Example ncsdfc Command Lines</u> .....	911
<u>ncshell</u> .....	912
<u>ncshell Command Syntax</u> .....	913
<u>ncshell Command Options</u> .....	914
<u>The Foreign Attribute</u> .....	919
<u>Importing LMSFI Models into VHDL</u> .....	920
<u>Importing SWIFT Models into VHDL</u> .....	922
<u>Importing FMI Models into VHDL</u> .....	924
<u>ncsuffix</u> .....	925
<u>ncsuffix Command Syntax</u> .....	925
<u>ncsuffix Command Options</u> .....	925
<u>Example ncsuffix Command Lines</u> .....	927
<u>ncupdate</u> .....	928
<u>ncupdate Command Syntax</u> .....	928
<u>ncupdate Command Options</u> .....	929
<u>Example ncupdate Command Lines</u> .....	933
<u>shellgen</u> .....	934
<u>shellgen Command Syntax</u> .....	935
<u>shellgen Command Options</u> .....	935

## NC-Verilog Simulator Help

---

<u>shellgen Command Example</u>	.....	939
<b><u>19</u></b>		
<u>The Programming Language Interface (PLI)</u>	.....	941
<b><u>20</u></b>		
<u>Importing Foreign Models</u>	.....	943
<u>The SmartModel SWIFT Interface</u>	.....	944
<u>Using the SmartModel SWIFT Interface with NC-Verilog</u>	.....	944
<u>Integrating SmartModel Library Models with NC-Verilog on UNIX</u>	.....	944
<u>Integrating SmartModel Library Models with NC-Verilog on Windows NT</u>	.....	946
<u>Running SmartModel Library Models with NC-Verilog</u>	.....	947
<u>The Hardware Modeling Interface (LMSI)</u>	.....	948
<u>Using LMG Hardware Models with NC-Verilog</u>	.....	948
<u>Integrating the LMG Hardware Modeling Interface with NC-Verilog</u>	.....	949
<u>Running LMG Hardware Models with NC-Verilog</u>	.....	950
<u>Specifying the Delay Mode for LMG Hardware Models</u>	.....	950
<u>The Open Model Interface (OMI)</u>	.....	952
<u>Integrating OMI Models</u>	.....	953
<u>Generating a Model Shell</u>	.....	954
<u>Integrating an OMI Model into a Verilog Design</u>	.....	955
<u>Integrating an OMI Model into a VHDL Design</u>	.....	959
<u>Modifying a Model Shell</u>	.....	962
<u>Simulating a Design With Imported OMI Models</u>	.....	963
<u>Simulating OMI Models Controlled by C++ Model Managers</u>	.....	966
<b><u>21</u></b>		
<u>Cosimulation with NC-Verilog and Quickturn</u>	.....	967
<u>Cosimulation Software Overview</u>	.....	967
<u>Setting Up the Simulator for Cosimulation</u>	.....	969
<u>Accessing Quickturn Integration</u>	.....	969
<u>Creating the Gate-Level Netlist</u>	.....	969
<u>Generating a Quickturn Emulator Database and a Pin Map</u>	.....	970
<u>Generating the Simulation Shell and Modifying the Testbench</u>	.....	970

## NC-Verilog Simulator Help

---

<u>Cadence Model Manager for Quickturn Command-Line Plus Options</u>	970
<u>Quickturn Modes for the qt mode Option</u>	971
<u>Specifying Cadence Model Manager for Quickturn Options at Simulation Time</u>	972
<u>omi Command</u>	973
<u>\$omiCommand System Task</u>	973
<u>Simulating a Model with NC-Verilog and Quickturn</u>	974
<u>Restrictions and Limitations</u>	975

### A

<u>Basics of Tcl</u>	976
<u>Overview</u>	976
<u>Tcl Basics</u>	977
<u>Tcl Variables</u>	977
<u>Variable Substitution</u>	977
<u>Tcl Commands</u>	977
<u>Command Substitution</u>	979
<u>Backslash Substitution</u>	979
<u>Value Substitution</u>	979
<u>Quoting Words in a Command</u>	980
<u>Extensions to Tcl</u>	982
<u>Expression Evaluation</u>	982
<u>Verilog Expressions</u>	982
<u>VHDL Expressions</u>	986
<u>Tcl Functions for Type Conversion</u>	991
<u>Enabling Tk in the NC-Verilog Simulator</u>	993

### B

<u>SDF File Syntax</u>	994
<u>Overview</u>	995
<u>SDF File Conventions</u>	996
<u>Identifiers</u>	996
<u>Operator Precedence</u>	999
<u>OVI Standard 3.0 SDF Keywords</u>	999
<u>SDF File Keyword Constructs</u>	1001
<u>DELAYFILE Keyword</u>	1001

## NC-Verilog Simulator Help

---

<u>CELL Keyword and Constructs</u>	1003
<u>CELLTYPE Keyword</u>	1004
<u>INSTANCE Keyword</u>	1004
<u>INCLUDE Keyword</u>	1005
<u>DELAY Keyword and Constructs</u>	1006
<u>ABSOLUTE Keyword</u>	1007
<u>INCREMENT Keyword</u>	1008
<u>IOPATH Keyword</u>	1009
<u>COND Keyword</u>	1012
<u>CONDELSE Keyword</u>	1013
<u>RETAIN Keyword</u>	1013
<u>PORT Keyword</u>	1014
<u>INTERCONNECT Keyword</u>	1016
<u>NETDELAY Keyword</u>	1018
<u>DEVICE Keyword</u>	1021
<u>PATHPULSE Keyword</u>	1023
<u>PATHPULSEPERCENT Keyword</u>	1024
<u>TIMINGCHECK Keyword and Constructs</u>	1025
<u>COND Keyword</u>	1026
<u>SETUP Keyword</u>	1027
<u>HOLD Keyword</u>	1028
<u>SETUPHOLD Keyword</u>	1029
<u>RECOVERY Keyword</u>	1030
<u>REMOVAL Keyword</u>	1032
<u>RECREM Keyword</u>	1033
<u>SKEW Keyword</u>	1034
<u>WIDTH Keyword</u>	1035
<u>PERIOD Keyword</u>	1036
<u>NOCHANGE Keyword</u>	1037
<u>TIMINGENV Keyword and Constructs</u>	1038
<u>PATHCONSTRAINT Keyword</u>	1039
<u>PERIODCONSTRAINT Keyword</u>	1041
<u>SKEWCONSTRAINT Keyword</u>	1042
<u>SUM Keyword</u>	1043
<u>DIFF Keyword</u>	1044
<u>ARRIVAL Keyword</u>	1045

## NC-Verilog Simulator Help

---

<u>DEPARTURE Keyword</u>	1046
<u>SLACK Keyword</u>	1047
<u>WAVEFORM Keyword</u>	1048
<u>OVI SDF Specification Version Differences</u>	1050
<u>SDF Version 1.* Constructs</u>	1050
<u>SDF Version 2.* Constructs</u>	1050
<u>SDF Version 3.* Constructs</u>	1050
<u>SDF File Examples</u>	1051
<u>Example 1</u>	1051
<u>Example 2</u>	1053
<u>Example 3</u>	1054

## C

<u>System Task Support in the NC-Verilog Simulator</u>	1055
<u>Example Tcl Script for \$countdrivers</u>	1065
<u>Example Tcl Script for \$display</u>	1066
<u>The \$readmemb and \$readmemh System Tasks</u>	1069
<u>Example Tcl Script for \$readmem</u>	1070
<u>Example Tcl Script for \$reset_count</u>	1072
<u>Example Tcl Script for \$reset_value</u>	1073
<u>Example PLI Routine for \$test\$plusargs</u>	1073

## NC-Verilog Simulator Help

---

<u>Example Tcl Script for \$showscopes</u>	1074
 <b>D</b>	
<u>Code Examples</u>	1076
<u>harddrive</u>	1077
<u>harddrive1</u>	1079
<u>shortdrive</u>	1081
<u>board</u>	1083
<u>count</u>	1085
<u>adder</u>	1087
<u>register</u>	1089
<u>top</u>	1092
<u>drink_machine</u>	1094
<u>drivers.vhd</u>	1107
<u>Verilog-VHDL-Verilog Sandwich</u>	1108
 <u>Glossary</u>	1113
 <u>Index</u>	1119

---

# Overview of the Cadence® NC-Verilog® Simulator

---

The Cadence® NC-Verilog® simulator is a Verilog digital logic simulator that combines the high-performance of native compiled code simulation with the accuracy, flexibility, and debugging capabilities of event-driven simulation. The NC-Verilog simulator is based on Cadence's Interleaved Native Compiled Code Architecture (INCA).

This chapter contains the following sections:

- [Native Compiled Code](#)
- [The Interleaved Native Compiled Code \(INCA\) Architecture](#)
- [The Cadence Desktop Simulators](#)
- [Language Support](#)
- [Memory Requirements](#)
- [Setting Up Your Design Environment](#)
- [Running the NC-Verilog Simulator](#)
- [Simulator Library Databases](#)

## Native Compiled Code

Native compiled code (NCC) is a software execution technique that provides a high-performance solution to the simulation performance bottleneck. In an NCC simulator, a parser produces an intermediate representation of the input source text. This intermediate representation is then processed by a code generator that produces relocatable machine code that runs directly on the host processor.

The NCC approach to simulation has several benefits over interpreted and compiled code techniques:

- Improved throughput, because the intermediate translation steps required by interpreted and compiled code simulators are bypassed.
- Significantly reduced time in setting up the simulation run because the use of the C compiler is avoided.
- More efficient use of memory.

NCC is the only technique that is optimal for use throughout the entire design cycle. It offers both fast design change turnaround, which is critical early in the design cycle, and a combination of fast simulation run time with the accuracy of full functional simulation, which is essential later in the process.

## The Interleaved Native Compiled Code (INCA) Architecture

The Interleaved Native Compiled Code Architecture (INCA) is an extension of the Native Compiled Code (NCC) approach to software execution.

The NCC approach to simulation addresses the performance challenge of a single-simulation strategy. However, many new factors are rapidly making single-language, event-driven, HDL-based simulation ineffective. These include:

- Increased design sizes that require a mixture of behavioral, rtl, and gate-level simulation for verification
- Intellectual property block usage and/or embedded block reuse that necessitate multilanguage environments
- Asynchronous, critical-path timing accuracy requirements that increase the emphasis on mixed-signal simulation
- Synchronous design simulation performance requirements that increase the emphasis on cycle simulation

INCA provides the performance of a single-engine NCC simulator with the flexibility to implement a multisimulation strategy. With INCA, a variety of verification disciplines can be supported, including:

- Multiple language (Verilog, VHDL, proprietary)
- Multiple levels (behavioral, rtl, gate)
- Multiple paradigms (event-driven, cycle-based)
- Mixed signal (digital, analog)

## **NC-Verilog Simulator Help**

### Overview of the Cadence® NC-Verilog® Simulator

---

With INCA, all the supported simulation styles leverage a single high-performance engine. Optimizing compilers for each input language or format create a sequence of instructions that are interleaved to create a single, contiguous code stream. This code stream is effectively a custom-built engine for the specific blend of simulation languages or techniques represented by a particular design.

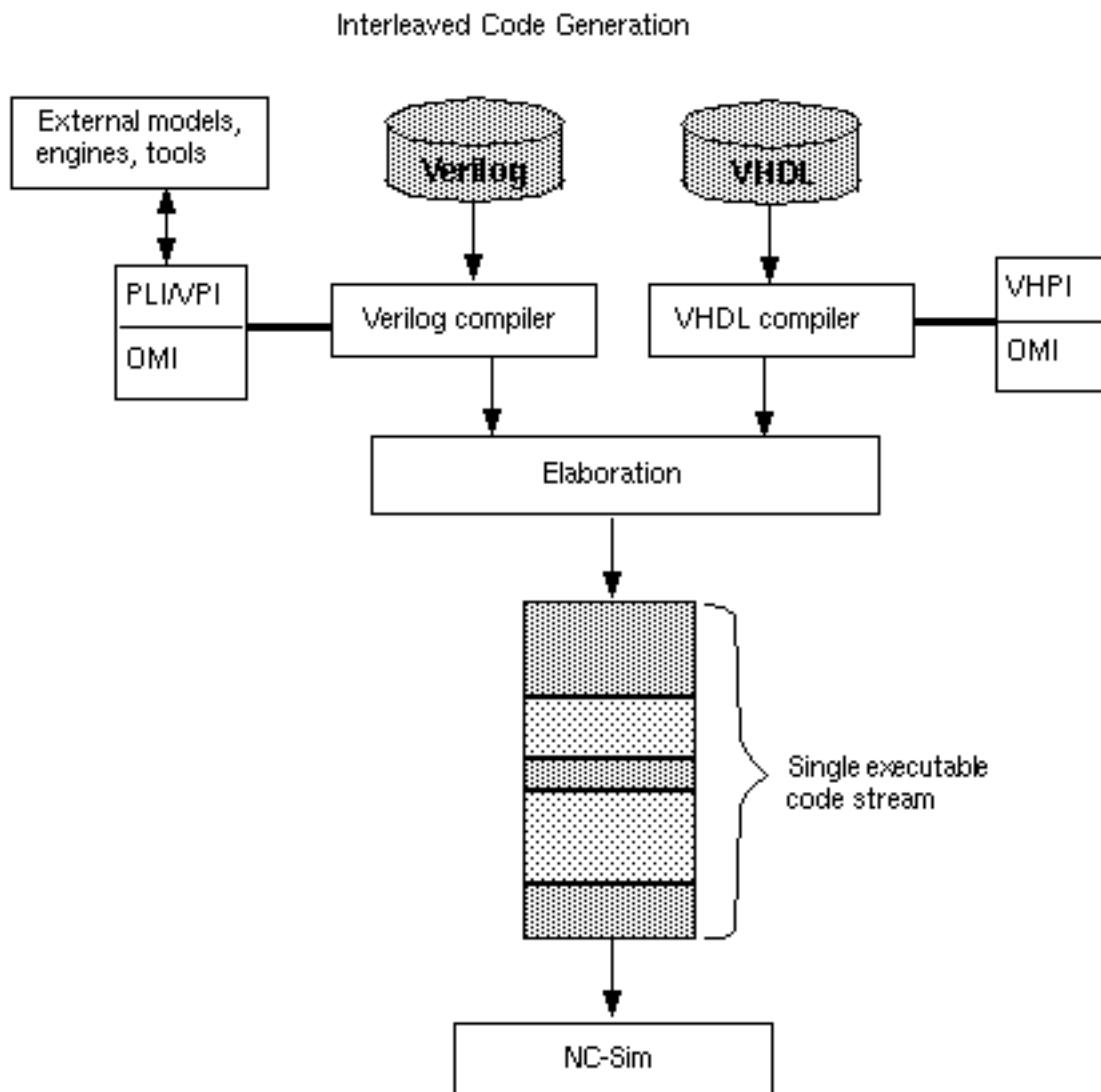
For example, in a Verilog/VHDL configuration, both the Verilog and VHDL compilers are used to generate code for the Verilog and VHDL portions of the design, respectively. During an elaboration process similar to the linking used in computer programming, the Verilog and VHDL code segments are combined into a single code stream. This single executable is then directly executed by the host processor.

This approach allows completely transparent mixed language, mixed-level, and mixed cycle-event simulations. It also lays the foundation for mixed signal simulations.

## NC-Verilog Simulator Help

### Overview of the Cadence® NC-Verilog® Simulator

The following figure illustrates the INCA mixed-language simulation flow:



## The Cadence Desktop Simulators

The Cadence Verilog Desktop, VHDL Desktop, and NC-Sim Desktop simulator products are low-cost, lower-performance versions of the full-performance NC-Verilog, NC-VHDL, and NC-Sim simulators. The Desktop products are available only on Windows NT and Windows2000 platforms.

The features and functionality of the Desktop products are the same as the full-performance simulators with the following differences:

- The Desktop has reduced performance.
- Some debug features that are off by default for the full-performance simulators are always on for the Desktop products.
- A few tools that are available with the full-performance simulators are not available on the Desktop because these tools have not yet been ported to Windows. These tools include:
  - Comparescan (database comparison)
  - NCBrowse (log file browser)
  - Code Coverage
  - HAL (HDL analysis and lint checker)
  - The Schematic Tracer and the Cycle View in SimVision

The Desktop simulators include a graphical user interface called NCLaunch for launching the tools, and the SimVision analysis environment. You can invoke NCLaunch from the *Start* menu.

Because the features of the Desktop simulators are the same as the features found in the full-performance NC simulators, no separate documentation is provided for the Desktop simulators. Use the documentation contained in this manual for both NC-Verilog and the Verilog Desktop.

For information on the VHDL Desktop simulator, use the *NC-VHDL Simulator Help*.

## Language Support

The NC-Verilog simulator is compliant with:

- The IEEE 1364 standard described in *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language* (IEEE Std 1364), published by the IEEE.
- The OVI 2.0 description of the language described in the *OVI Verilog Hardware Description Language Reference Manual*, Version 2.0, published by OVI.
- The Verilog-XL implementation of the Verilog language described in the *Verilog-XL Reference Manual*.

You can use the `-ieee1364` command-line option when you invoke the `ncvlog` compiler and the `ncelab` elaborator to check your code for compatibility with the IEEE standard.

## Memory Requirements

As with any simulator, memory requirements for the NC-Verilog simulator are highly dependent on the size of the design. In order to achieve the highest performance possible, you must have enough memory to compile and elaborate the design efficiently, and, during the actual simulation phase, you should have enough memory so that the design resides in physical memory.

For RTL designs, a minimum of 64 Mb is required for both building and simulating the design.

For a gate-level design of about 150K gates, 128 Mb is recommended for optimal build time. For simulation, 64 Mb should be sufficient.

## Setting Up Your Design Environment

In the NC-Verilog simulator, compiled objects (Verilog modules, macromodules, and UDPs) and other derived data are stored in libraries. The library structure is organized around a Library.Cell:View approach, where:

- A library relates to a specific design or to a reference library.
- Cells relate to specific modules or building blocks of the design.
- Views relate to different representations of the building blocks.

See “[The Library.Cell:View Approach](#)” on page 109 for details on NC-Verilog’s library system.

Each library has a logical name and is represented by a unique directory. When you compile and elaborate a design, all of the internal representations of cells and views that are required by the simulator are contained in a single file stored in the library directory.

To run the NC-Verilog simulator, two setup files are required:

- A `cds.lib` file. This file contains statements that define your libraries and that map logical library names to physical directory paths. See “[The cds.lib File](#)” on page 110 for details.
- An `hdl.var` file. This file defines which library is the work library. The `hdl.var` file also can contain definitions of other variables that determine how your design environment is configured, control the operation of NC-Verilog tools, and specify the locations of support files and invocation scripts. See “[The hdl.var File](#)” on page 118 for details on the `hdl.var` file.

**Note:** If you run the NC-Verilog simulator with the `ncverilog` command (see “[Running the NC-Verilog Simulator](#)” on page 31) or if you run the `ncprep` utility, these two files are created for you automatically.

You can have more than one `cds.lib` or `hdl.var` file.

All NC-Verilog tools and utilities that require a `cds.lib` and `hdl.var` file use the same search mechanism to find the files. This search mechanism is described in “[The setup.loc File](#)” on page 131. The first file that is found is used.

You can write a `setup.loc` file to change the directories to search or to change the order of precedence to use when searching for the `cds.lib` and `hdl.var` files. Each tool and utility also includes command-line options (`-cdslib` and `-hdlvar`) that let you specify which setup files to use directly on the command line.

## Running the NC-Verilog Simulator

There are two ways to run the NC-Verilog simulator:

- Single-step invocation

In this way of running the simulator, you issue one command, the `ncverilog` command. This command invokes a parser called `ncvlog` and an elaborator called `ncelab` to build the model, and then invokes the `ncsim` simulator to simulate the model.

- Multi-step invocation

In this way of running the simulator, you invoke `ncvlog`, `ncelab`, and `ncsim` separately.

The invocation method that you use depends on a variety of factors, such as your current simulation environment, whether or not you want to modify this environment, whether or not users are using both the Verilog-XL and NC-Verilog simulators, and, perhaps most importantly, whether or not you want the NC-Verilog simulator to handle `-y` and `-v` technology libraries exactly the same way that Verilog-XL handles those libraries.

In either invocation model, the build and simulation steps are the same and serve essentially the same purpose. The cell binding mechanism is the major difference between the two invocation methods.

- `ncvlog` analyzes and compiles your Verilog source. This tool performs syntactic checking on the HDL design unit(s) (modules, macromodules, or UDPs) in the input source file(s) and generates an intermediate representation for each HDL design unit. These intermediate representations are stored in a single file contained in the library directory. This library database file is called:

## NC-Verilog Simulator Help

### Overview of the Cadence® NC-Verilog® Simulator

`inca.architecture.lib_version.pak`

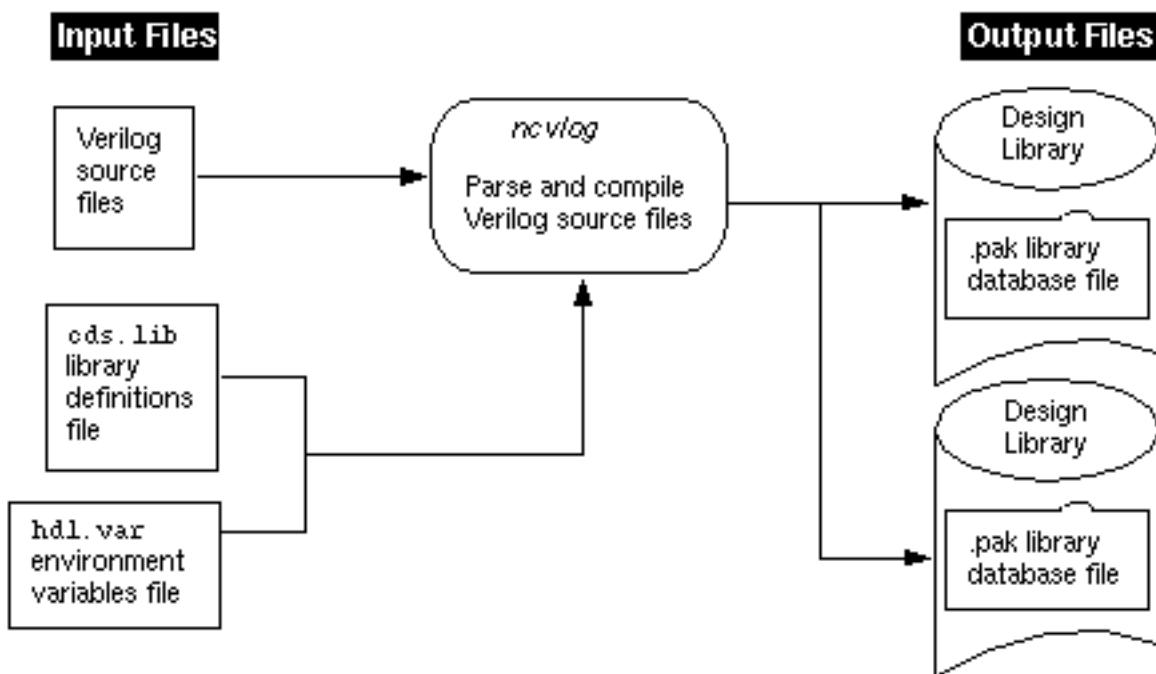
For example, on Solaris, the name of the library database file is similar to the following:

`inca.sun4v.091.pak`

See “[Simulator Library Databases](#)” on page 38 for more information on library databases.

In single-step invocation, *ncvlog* creates a binding list that the elaborator uses, and any change that you make to a source file causes that binding list to be regenerated.

The following figure shows the inputs and outputs of *ncvlog*.



See [Chapter 7, “Compiling Verilog Source Files with ncvlog.”](#) for more information on compiling with *ncvlog*.

- *ncelab* elaborates the design hierarchy that defines the model. The elaborator takes as input the Library.Cell:View name of the top-level HDL design unit(s). It then constructs a design hierarchy based on the instantiation and configuration information in the design, establishes connectivity, and computes the initial values for all of the objects in the design.

The code generator, which produces the machine code, runs as a subprocess of elaboration.

If *ncelab* does not find any errors, it produces a simulation snapshot. The snapshot contains the simulation data at simulation time 0, and is the input to the *ncsim* simulator.

The machine code and the snapshot are both stored in the library database file, along with the intermediate objects that are the result of compilation.

By default, the elaborator generates a snapshot in which simulation constructs are marked as having no read, write, or connectivity access. By limiting access to simulation objects, the elaborator can perform several optimizations that greatly increase performance.

When you are running simulations in “regression” mode, the default access level is the obvious choice. However, if you run the simulator in this mode, you will not be able to access objects from a point outside the HDL code. For example, you cannot probe objects that do not have read access, and waveforms cannot be generated for these objects.

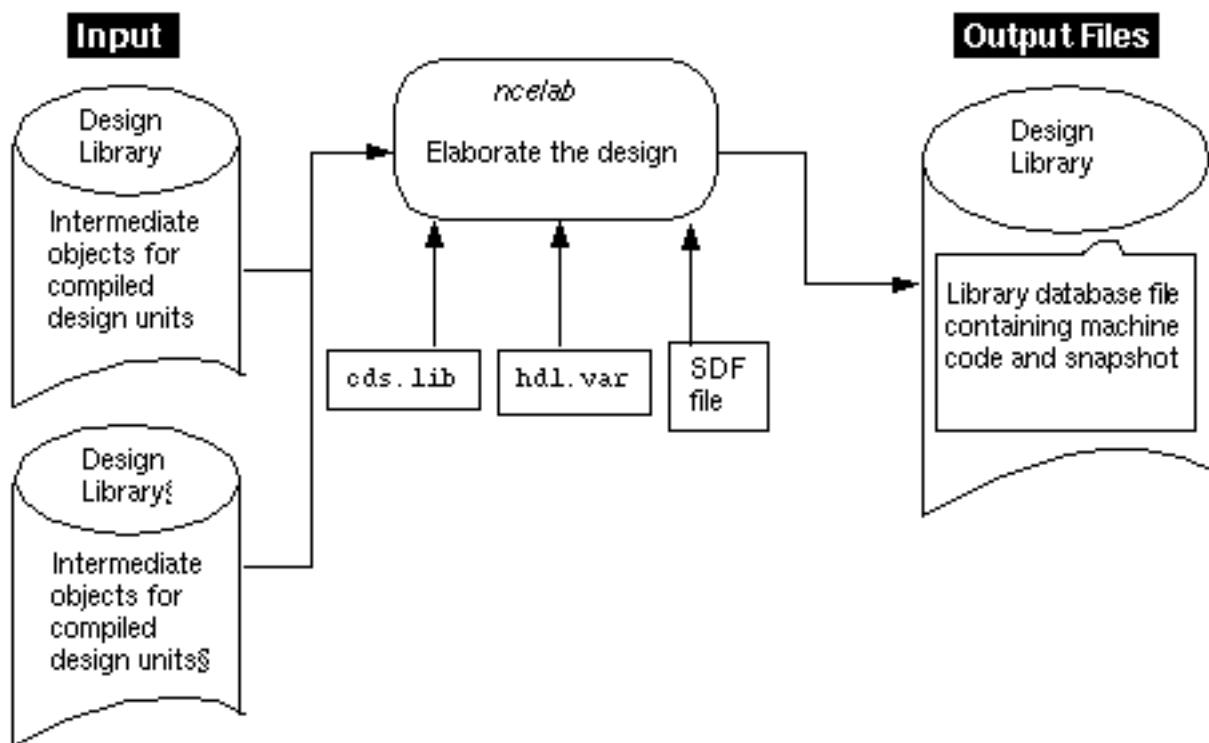
If you want to run the simulation in debug mode, with access to simulation objects, use the `-access` option (`+ncaccess+` for *ncverilog*) to enable the different kinds of access to simulation objects. You can also specify the access capability for particular instances and for parts of a design by including an access file with the elaborator `-afile` option (`+ncafile+` for *ncverilog*).

See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for more information on running the NC-Verilog simulator in regression mode versus running the simulator in debug mode.

## NC-Verilog Simulator Help

### Overview of the Cadence® NC-Verilog® Simulator

The following figure shows the inputs and outputs of *ncelab*.



In multi-step invocation, the elaborator makes all binding decisions. In single-step invocation, the elaborator uses the binding list that *ncvlog* generates.

See [Chapter 8, “Elaborating the Design with ncelab.”](#) for more information on elaborating with *ncelab*.

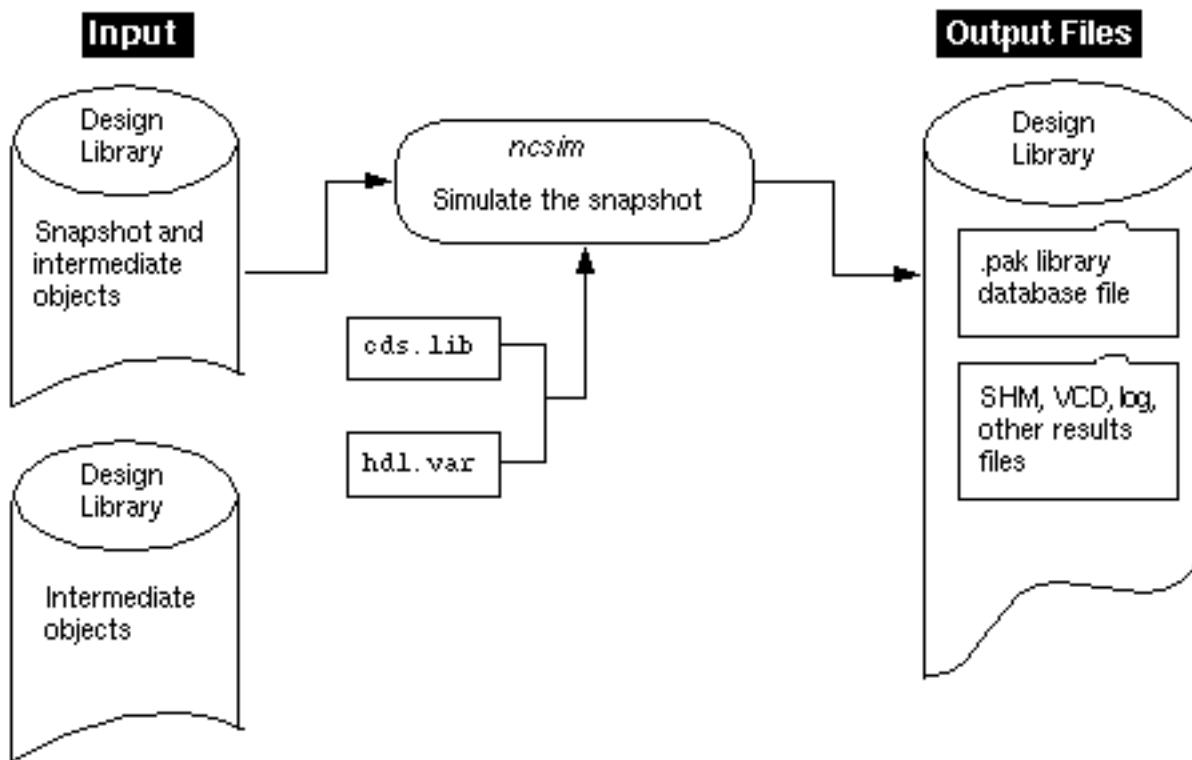
- *ncsim* simulates Verilog using the native instruction streams to execute the dynamic behavior of the design.

The simulator loads the snapshot generated by the elaborator, as well as other objects that the compiler and elaborator generate that are referenced by the snapshot. The simulator may also load HDL source files, script files, and other data files as needed (via \$read\* tasks or textio). *ncsim* can generate a log file, an SHM or VCD database, and other results files.

## NC-Verilog Simulator Help

### Overview of the Cadence® NC-Verilog® Simulator

The following figure shows the inputs and outputs of *ncsim*.



See [Chapter 9, “Simulating Your Design with \*ncsim\*,”](#) for details on simulating with *ncsim*.

You can invoke the *ncsim* simulator:

- In noninteractive mode, so that simulation runs after initialization without waiting for your command input.
- In interactive mode, so that the simulator stops at time 0.

You can also invoke the simulator with the SimVision analysis environment. The SimVision analysis environment is a comprehensive debug environment that consists of:

- A main window, called SimControl, in which you can view your source and perform a wide variety of debug operations.
- Advanced debug tools that you can access from SimControl. These tools include:
  - The Navigator, which lets you view your current design hierarchy in a graphical tree representation and as a list of objects with their current simulation values and declarations.
  - The Watch Window, which lets you select and then watch signal value changes.

## NC-Verilog Simulator Help

### Overview of the Cadence® NC-Verilog® Simulator

---

- ❑ The Signal Flow Browser, which lets you trace backwards through a design from a signal that has a questionable value to where a signal first diverges from the expected behavior.
- ❑ The Cycle View , which lets you step through the simulation cycle to debug delta cycle bugs in the design.
- ❑ The Schematic Window, which lets you view RTL models in a schematic form. The Schematic Window can display the schematic for an entire module that you select in the Navigator, or it can display a signal and its connected logic if you select a signal in the Source Browser or other SimControl tool.

In the current release, the Schematic Window is available only for pure Verilog designs. This tool is not available in the Verilog desktop simulator.

The SimVision analysis environment also includes [SimVision Waveform Viewer](#), a waveform display tool, and [Comparescan](#), a tool that lets you compare results stored in SHM (SST2) or VCD databases.

The SimVision debug environment can also be invoked in post processing mode. The Post Processing Environment (PPE) lets you analyze simulation results stored in an SHM database and debug your design without using a simulator license. The PPE gives you access to all SimVision tools available in interactive mode. All tools work together as they do in interactive mode, and the features of each tool are virtually identical in both modes.

See the [Cadence SimVision Analysis Environment User Guide](#) for details on using the SimVision analysis environment.

Because the NC-Verilog simulator is a compiled code simulator that does not contain an interpreter, and because *ncsim* must be able to display and manipulate mixed-language constructs, you cannot enter Verilog commands at the command-line prompt. The NC-Verilog simulator supports a set of Tool Command Language (Tcl) commands for interactive debugging. See [Chapter 12, “Using the Tcl Command-Line Interface.”](#) for a list of interactive commands.

**Note:** Remember that if you run *ncelab* in the default (regression) mode to elaborate the design, simulation objects are tagged as having no read, write, or connectivity access. A warning or error message is displayed if you execute a Tcl command that requires read or write access.

You can use Tk with the NC-Verilog simulator. Tk is a toolkit for the X Windows System that extends the Tcl facilities with commands that you can use to build user interfaces, so that you can develop Motif-like user interfaces by writing Tcl scripts instead of writing C code. Tk is not shipped with the simulator. However, the required shared library and the library of Tcl script files is available on the internet. See [“Enabling Tk in the NC-Verilog Simulator”](#) on page 993 for instructions on enabling Tk in the NC-Verilog simulator.

## Single-Step Invocation With `ncverilog`

The single-step invocation model is intended primarily for Verilog-XL users who want to improve the simulation performance of designs that are already working in a Verilog-XL environment and for those users who need to switch back and forth between the two simulators.

`ncverilog` lets you run the NC-Verilog simulator in exactly the same way that you run Verilog-XL. You invoke the simulator with a single command, `ncverilog`. The command-line options and arguments are the same options and arguments that you pass to Verilog-XL. For example, if you run Verilog-XL with the following command:

```
% verilog -f verilog.args
```

You run the NC-Verilog simulator with this command:

```
% ncverilog -f verilog.args
```

Besides Verilog-XL command-line options, you can also include `ncvlog`, `ncelab`, and `ncsim` options on the `ncverilog` command line in the form of plus options. There are also some plus options that are specific to the `ncverilog` command.

Running the simulator with the `ncverilog` command automatically creates everything you need to run the simulator, including all directories, libraries, a `cds.lib` file, and an `hdl.var` file. The simulator then translates all applicable Verilog-XL options into options for the NC-Verilog simulator and then invokes the parser and compiler (`ncvlog`), the elaborator (`ncelab`), and the simulator (`ncsim`) sequentially to simulate the design.

Running the NC-Verilog simulator with the `ncverilog` command is recommended for designs that are already working in a Verilog-XL environment and for designers coming from a Verilog-XL background. The three primary reasons for this recommendation are:

- Convenience. The `ncverilog` use model matches that of Verilog-XL. You can use the same command files and command-line arguments for both simulators. This becomes especially important if you need to switch back and forth between the two simulators.
- Ease of use. No setup is required for single-step invocation. All you do is invoke the tool and its options on the source files. Using `ncverilog` improves the simulation performance of designs that are already working in your Verilog-XL environment without requiring you to modify your simulation work flow or design environment. This use model also lets you evaluate the NC-Verilog simulator using an existing design that simulates in Verilog-XL.
- Search order. `ncverilog` mimics the search order of Verilog-XL when it binds instances. The single-step invocation model understands `-y` and `-v` technology libraries and manages them within the parser, as does Verilog-XL. `ncverilog` uses the same library search order that Verilog-XL uses, duplicates the binding rules of Verilog-XL, and propagates macros the same way that Verilog-XL does.

See [Chapter 4, “Running NC-Verilog with the ncverilog Command,”](#) for more information on *ncverilog*.

## **Multi-Step Invocation (Library-Based Mode)**

You can run the NC-Verilog simulator by executing the three main tools in succession. Each tool is invoked with its own command line and arguments.

Two setup files are required: a library definition file (`cds.lib`) and a tool environment variables file (`hdl.var`). While rudimentary `cds.lib` and `hdl.var` files can be used, these files are the main means of manipulating the environment and can become quite complex.

This way of running the NC-Verilog simulator is recommended for designs that are organized in a library-based system as opposed to a file-based system, such as that used by Verilog-XL. It should also be used if your environment does not depend on being able to switch back and forth between NC-Verilog and Verilog-XL.

The multi-step invocation method:

- Provides more flexibility and more control over the placement and reuse of intermediate files.
- Uses a simpler set of binding rules than those used in single-step invocation, which reproduces the Verilog-XL binding mechanism. Binding is more predictable and manageable.
- Provides finer control over the update mechanisms.
- Provides better incremental recompile performance for designs that continually rescan a directory or several directories or files. This behavior is eliminated if the design is organized in a library-based system, and is therefore much more efficient.

## **Simulator Library Databases**

When you compile and elaborate a design, all intermediate objects are stored in a single file in the design library. This library database file is called `inca.architecture.lib_version.pak`. For example, on Solaris, the name of the library database file looks like the following:

`inca.sun4v.091.pak`

Library database files are read/write by default. You can use the `ncpack` utility to change the properties of a database to make it read-only or add-only.

## NC-Verilog Simulator Help

### Overview of the Cadence® NC-Verilog® Simulator

---

In virtually all cases, you can treat the contents of a library system as a black box. If, however, you need to list the objects contained in the library system, the *nc/s* utility provides this visibility.

A file locking mechanism is used to manage multiple processes that might need to read or modify the contents of a library at one time. If a process cannot get a required lock, a warning is issued, and the process tries again a short time later. If a process cannot get a lock after approximately one hour, the process times out and exits.

The following two messages are examples of the warning messages that are generated by the file locking mechanism.

```
ncvlog: *W,DLWTLK: Waiting for a read lock on library 'alt_max2'.  
ncvhdl_cg: *W,DLWTLK: Waiting for a write lock on library 'worklib'.
```

In rare cases, file locking can result in a real deadlock situation in which neither process can proceed because it is waiting for the other process to release a lock. For example, some processes suspended with a CTRL/Z retain their locks when suspended (an *ncelab* process, for example). In these cases, you must terminate a process manually. You can use the *ncpack -unlock* command to do this.

A signal handling mechanism ensures that any unexpected event, such as a Ctrl/C, flushes the database to the disk to avoid corruption of the library. However, conditions such as terminating a process with *kill -9* or a power failure can corrupt a library database. In these cases, delete the library database file and rebuild.

The following example shows the message that is generated when the library has been corrupted.

```
ncvlog: *F,DLPAKC: Packed library for alt_max2 is corrupt, please remove  
../alt_max2/inca.sun4v.091.pak.
```

The following operating systems impose a two gigabyte limit on the size of a file:

- Windows NT
- Linux Red Hat 6.1 and 6.2

If a library database exceeds this limit, you will not be able to add objects to the database. To work around this limitation, you can create multiple libraries, each of which is smaller than the two gigabyte limit.

## **NC-Verilog Simulator Help**

### Overview of the Cadence® NC-Verilog® Simulator

---

The following operating systems do not impose the two gigabyte limit on the size of a file:

- Solaris 2.7
- HP-UX 11.0

---

## Getting Help

---

This chapter contains the following sections:

- [About Online Help](#)
- [Getting Help on Commands to Run Tools](#)
- [Getting Help on Simulator Commands](#)
- [Getting Help on Tool Messages](#)
- [Related Manuals and Specifications and Other Documentation](#)

### About Online Help

Documentation for the NC-Verilog simulator is provided in HTML and in PDF format.

The online documentation system consists of:

- The Cadence documentation window.

This window lets you find and open any of the books shipped with the products that you ordered. You can list books by product name, by product family, or by document type. For example, you can list all manuals, all Product Notes documents, or all Known Problems and Solutions documents. When you select a document, it is opened in your Web browser. The system automatically starts the browser, if necessary.

- Manuals in HTML format.

Each HTML document has both hyperlinked cross-references and a tool bar with buttons that let you navigate through the documentation system. Using the buttons on the tool bar, you can redisplay the documentation window, move forward and backward through chapters, display the Table of Contents, open a PDF file for printing, or open the search page. You can also send an e-mail directly to Cadence publications with comments about the document that you are viewing.

- A PDF (Portable Document Format) file for each document so that you can print the entire document or sections of a document.
- A powerful search tool that lets you search for information in documents for a product family or for specific products. You can also search individual documents.

Click the *Help* button on the tool bar of any document to open the *Cadence Documentation User Guide*, which contains a complete description of how to use the online documentation system.

## **Invoking the Documentation System**

There are two ways to open the online help for the simulator:

- From the Cadence documentation window.

To invoke the Cadence documentation window on UNIX or Linux, use the `cdsdoc` command.

% `cdsdoc`

On Windows, you can:

- Open a command window and enter the `cdsdoc` command.
- Use Windows Explorer to open `your_install_dir\tools\bin` and then double-click `cdsdoc.exe`.

On the Cadence documentation window, click on a category name to show the documents in that category. Then double-click a manual title to load that manual into your web browser.

- From the Help menu on the graphical user interface.

If you are using a graphical user interface, such as NCLaunch or the SimVision analysis environment, pull down the *Help* menu and select the name of the online manual that you want to view. For example, if you are simulating a Verilog design, select *NC-Verilog Help* to open the online help for the NC-Verilog simulator. This displays the HTML file in your Web browser.

Help can also be accessed from forms on the graphical interface. Click on the *Help* button on the bottom right of the form to get online help.

To open the Cadence documentation window, click the Library button at the top of any document displayed in the browser.

## Printing Documents

The *View/Print PDF* button opens a PDF file in Adobe Acrobat® Reader. You can print the entire document or print a section of the document by specifying a range of page numbers.

A hypertexted list of contents (a "bookmark" list) is available for navigating the print document online. Hypertext cross-reference links in the HTML version are reformatted to include page numbers in the printed copy, so that you can find the referenced page easily when reading the printed version.

## Searching Documents

The built-in search mechanism lets you search various groupings of books or individual books. You can:

- Search all installed documents.
- Search all documents for specific products or product families.
- Search one or more specific books.

The Search tool lets you perform many different types of full-text search queries. You can search for text phrases or exact words, use Boolean AND, OR, or NOT operators, use special operators such as CASE (for case-sensitive searches) or NEAR (to search for words near each other), or use wildcard characters for substitution.

## Getting Help on Commands to Run Tools

You can display a list of options for any of the simulator tools and utilities by typing the tool or utility name followed by the *-help* option.

The *-help* option displays a list of the command options for the specified tool with a brief description of each option.

Syntax:

```
% tool_name -help
```

Examples:

```
% ncvlog -help  
% ncvhdl -help  
% ncelab -help  
% ncsim -help  
% ncupdate -help
```

## Getting Help on Simulator Commands

To get help on simulator (*ncsim*) commands:

- Use the `help` command.

```
ncsim> help [help_options] [command | all [command_options]]
```

In addition to the set of interactive commands implemented for the simulator, the `help` command also displays help on standard Tcl commands. Only basic information is provided for these commands. See the following Web sites for information on Tcl commands and for other information related to Tcl/Tk:

```
http://www.elf.org  
http://dev.scriptics.com/  
http://www.tcltk.com/
```

- Use the command reference section in this online help. See [Chapter 12, “Using the Tcl Command-Line Interface,”](#)
- If you are using the SimVision analysis environment, use the Quick Help feature. Click on a menu, such as the File menu, to display the commands on that menu. Then position the pointer over a command. Press and hold the Shift key and click the middle mouse button to display help for that command.

Some commands pop up forms that you have to fill in. Click on the *Help* button on the form to get more information about that form.

## Getting Help on Tool Messages

Use the *nchelp* utility to display extended help on the brief messages generated by the compiler, elaborator, and simulator.

Syntax:

```
% nchelp [options] tool_name message_code
```

You can enter the *message\_code* argument in lowercase or in uppercase.

Examples:

```
% nchelp ncvlog BADCLP  
% nchelp ncvlog badclp  
% ncelp ncelab cuvwsp  
% ncelp ncsim NOSNAP
```

## Related Manuals and Specifications and Other Documentation

This section lists the names of relevant language specifications and books on the Verilog and VHDL hardware description languages.

### Verilog

- *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language* (IEEE Std 1364), IEEE, New York, October, 1996.
- *OVI Verilog Hardware Description Language Reference Manual*, Version 2.0
- [Verilog-XL Reference Manual](#)
- [PLI 1.0 User Guide and Reference](#)
- [VPI User Guide and Reference](#)
- Ousterhout, John K., *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, MA, 1994.

There are several books available that deal with the Verilog HDL. These include:

- Donald E. Thomas and Philip Moorby, *The Verilog Hardware Description Language*, Kluwer Academic Publishing, 1996.
- E. Sternheim, R. Singh and Y. Trivedi, *Digital Design With Verilog HDL*, Chapman & Hall, 1991.
- E. Sternheim, *Digital Design and Synthesis With Verilog HDL*, Automata Publishing Company, 1993.
- R. Madhavan, *Quick Reference for Verilog HDL*, Automata Publishing Company.
- J. Bhasker, *A Verilog HDL Primer*, Star Galaxy Press, 1997.
- Douglas J. Smith, *HDL Chip Design: A Practical Guide for Designing, Synthesizing and Simulating ASICs and FPGAs Using VHDL or Verilog*, Doone Publications, 1996.

- Samir Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, Prentice Hall, 1996.
- James M. Lee, *Verilog Quickstart*, Kluwer Academic Publishing, 1997.

## VHDL

- *IEEE Standard VHDL Language Reference Manual (IEEE Std 1076 - 1987)*, IEEE, New York, March 1988.
- *IEEE Standard VHDL Language Reference Manual (IEEE Std 1076 - 1993)*, IEEE, New York, June 1994.

There are several books available that deal with the VHDL language. These include:

- Douglas J. Smith, *HdI Chip Design: A Practical Guide for Designing, Synthesizing & Simulating Asics & Fpgas Using Vhdl or Verilog*, Doone Publications, June 1996.
- Douglas L. Perry, *VHDL 3rd Edition*, McGraw Hill Text, June 1998.
- Jayaram Bhasker, Jayram Bhaskar, *A Guide to Vhdl Syntax: Based on the New IEEE Std 1076-1993 (Innovative Technology)*, Prentice Hall Press, 1995.
- Peter J. Ashenden, *Designer's Guide to Vhdl*, Morgan Kaufman Publishers, December 1995.
- Stanley Mazor, Patricia Langstraat, *A Guide to Vhdl 2nd edition*, Kluwer Academic Publishers, August 1993.
- Andrew Rushton, *VHDL for Logic Synthesis: An Introductory Guide for Achieving Design Requirements*, McGraw-Hill Book Company, 1995.
- James R. Armstrong, F. Gail Gray, Gail Gray, *Structured Logic Design With Vhdl*, Prentice Hall Press, 1993.

## Other Documentation

A wealth of other documentation related to Cadence simulation products is available on the Web. These sites contain product information, datasheets, information on what's new in the latest release, application notes, white papers, information about Cadence services, such as training, customer support, and methodology services, and so on.

There are two primary sites:

## **NC-Verilog Simulator Help**

### **Getting Help**

---

- <http://ncsim.com>

This site is specifically designed for users who need to solve tough functional verification problems and would like to share their experiences with a broader community.

- <http://sourcelink.cadence.com>

SourceLink is a technical support service for Cadence software users. The service is available to all customers who have a software support services agreement.

---

## Using NCLaunch

---

NCLaunch is a graphical user interface that is integrated into the Cadence Interleaved Native Compiled Architecture (INCA). NCLaunch helps you to manage large design projects by presenting you with a unified view of the files and libraries in your design and by providing you with an easy and consistent way to configure and launch your Cadence simulation tools. NCLaunch provides easy access to all of the tools that you need to run a simulation.

You can use NCLaunch with any NC simulator, including the Desktop simulators.

On UNIX, invoke NCLaunch by typing the `nclaunch` command in a command window.

```
% nclaunch &
```

On Windows, invoke NCLaunch by selecting *Start—Programs—Cadence Design Systems—Affirma Design and Verification—NCLaunch*.

The NCLaunch tool consists of a single main window with two browsers that are integrated with the suite of NC tools. Both browsers simply display the information stored in your directories in a way that makes the information easy to interact with. The integrated tools are the compilers (*ncvhdl* and *ncvlog*), the elaborator (*nclab*), and the simulator (*ncsim*). NCLaunch also integrates other debug tools, such as SimVision Waveform Viewer, Comparescan, and NCBrowse, and utilities, such as the SDF compiler (*ncsdfc*), and *ncupdate*.

**Note:** Comparescan and NCBrowse are not currently available on Windows platforms.

NCLaunch consists of the following components:

- The File Browser

This browser, which appears on the left-hand side of the NCLaunch window, displays the files that make up your design. You can select files and then execute commands by selecting a command from a pulldown menu or popup menu, or by clicking on a button on the Tool Bar.

For example, you can select Verilog source files in the File Browser and then compile the files by selecting *Tools—Verilog Compiler*, by selecting *NCVlog* from the popup menu, or by simply clicking the *Launch Verilog Compiler* button on the Tool Bar. All tool

## NC-Verilog Simulator Help

### Using NCLaunch

---

options are available through the *Tools* pulldown menu. When you change the tool options, those options remain in effect until you change the options again. All option settings are saved when you exit NCLaunch and are reused the next time that you invoke the tool. This lets you save options and associate them with a specific design.

#### ■ The Library Browser

This browser, which appears on the right-hand side of the NCLaunch window, reads and displays the contents of a `cds.lib` file and its corresponding libraries. The `cds.lib` file and its libraries are displayed as a tree so that you can expand the libraries that you want to view and manipulate the cells and design units that make up the library. The Library Browser can display the libraries in either the packed library or 5.X format.

As with the File Browser, you can select objects in the Library Browser and then execute commands on these objects. For example, you can select a top-level design unit and then elaborate the design by selecting *Tools—Elaborator*, by selecting *NCElab* from the popup menu, or by clicking on the *Launch Elaborator* button on the Tool Bar.

#### ■ I/O Region

This area of the NCLaunch window, which appears beneath the browsers, lets you submit Tcl commands and view the output of running processes.

#### ■ The Status Bar

This is a small window that runs across the bottom of the NCLaunch window. A single line of text is displayed to inform you about various UI activities. When you move the mouse over a Tool Bar button or a menu item, the status bar area displays a message that describes the function of that item. This area also displays the number of selected items in the NCLaunch window.

See the [\*NCLaunch User Guide\*](#) for details on using NCLaunch.

---

## Running NC-Verilog with the ncverilog Command

---

This chapter contains the following sections:

- [Overview](#)
- [How ncverilog Works](#)
- [ncverilog Command Syntax](#)
- [ncverilog Command Options](#)
- [Plus Options for NC-Verilog Tools](#)
- [Verilog-XL Command-Line Options Translation](#)
- [Example ncverilog Run](#)
- [Mapping of Warning Messages](#)
- [Updating Design Changes](#)
- [Using +ncuid+ to Run in Regression Mode](#)
- [Using -R and -r to Simulate a Snapshot](#)
- [PLI Tasks](#)
- [SDF Annotation](#)
- [Troubleshooting](#)

## Overview

You can run the NC-Verilog simulator by issuing one command, the `ncverilog` command. This command spawns the *ncvlog* parser and the *ncelab* elaborator to build the model, and then spawns the *ncsim* simulator to simulate the model. This single-step invocation model is intended primarily for Verilog-XL users who want to improve the simulation performance of designs that are already working in an XL environment and for those who need to switch back and forth between the two simulators.

You invoke *ncverilog* with the `ncverilog` command followed by the Verilog-XL command-line arguments. For example, suppose you invoke Verilog-XL with an arguments file that contains all dash (-) and plus (+) options and all source files, as follows:

```
% verilog -f verilog.args
```

To run *ncverilog*, type:

```
% ncverilog -f verilog.args
```

Besides Verilog-XL command-line options, you can include *ncvlog*, *ncelab*, and *ncsim* options on the `ncverilog` command line. These tools, if run separately, take dash options of the form `-option`. Many of these options have a corresponding plus option that you can use on the `ncverilog` command. All of these options begin with `+nc`. For example, the `ncvlog -ieee1364` option has a corresponding `ncverilog +ncieee1364` option. See “[Plus Options for NC-Verilog Tools](#)” on page 66 for a list of these options.

There are also some plus options that are specific to the `ncverilog` command. See “[ncverilog Command Options](#)” on page 56.

Running *ncverilog* automatically creates everything you need to run the NC-Verilog simulator, including all directories, libraries, a `cds.lib` file, and an `hdl.var` file, if they don’t already exist. It then translates all applicable Verilog-XL options and then invokes *ncvlog*, *ncelab*, and *ncsim* sequentially to simulate the design.

*ncverilog* uses the same library search order that Verilog-XL uses, duplicates the binding rules of XL, and propagates macros the same way that XL does.

When you run *ncverilog*, the parser is invoked with the `-update` option by default. This option minimizes compile time and maximizes reuse of previously compiled objects.

By default, the elaborator generates a snapshot with simulation objects marked as having no read, write, or connectivity access. This increases the performance of the simulator for long regression test runs, but does not provide the access to objects that you need to debug a design. There are several *ncverilog* command-line options you can use to specify access to simulation objects:

## NC-Verilog Simulator Help

### Running NC-Verilog with the ncverilog Command

---

- +debug. This option turns on read access to all objects in the design. Read access is required for probing nets, regs and variables (including setting PLI callbacks) and getting the value of these objects. The `ncverilog +debug` option is translated to the `ncelab -access +r` option.
- +ncaccess+. Use this option to selectively turn on different kinds of access. Using `+ncaccess+` allows you to be specific about the type(s) of access you need for your debugging purposes. For example, if you need read access turned on so that the simulator can dump waveforms, you can specify read access only by using `+ncaccess+r`. If you need write access to objects so that you can deposit and force values, you can specify read and write access by using `+ncaccess+r+w`.
- +ncafile+access\_file. Use this option to specify an access file, which lets you set the visibility access for particular instances or portions of a design.

You can use the `+ncgenafile+access_filename` option to automatically generate an access file and then use the `+ncafile+` option in a subsequent run to use the access file.

See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for more information on specifying access to simulation objects.

If you want to queue simulation jobs so that they are run when licenses become available, make sure that the XL arguments file contains one of the license queueing options (`+licq*`). Any XL `+licq*` option is automatically translated to the NC-Verilog simulator license queueing option (`ncsim -licqueue`).

`ncverilog` command-line options can be specified in an `hdl.var` file with the `NCVERILOGOPTS` variable.

The NC-Verilog simulator command language is based on Tcl. You cannot use the `-i` option to specify an input file containing Verilog commands. To specify an input file containing Tcl commands, use the `+ncinput+filename` or `+tcl+filename` option.

See [Chapter 12, “Using the Tcl Command-Line Interface.”](#) for information on the Tcl commands.

## How ncverilog Works

This section summarizes how *ncverilog* works. The explanation assumes that you are just substituting the *ncverilog* command for the *verilog* command to run the simulation.

The first time you run the NC-Verilog simulator with the *ncverilog* command, it:

1. Creates a directory called `INCA_libs`.
2. Creates a work library called `INCA_libs/worklib`.
3. Creates a subdirectory called `INCA_libs/snap.nc`. The NC-Verilog simulator uses this directory as a scratch area to create files and pass them between tools.
4. Creates a file called `ncverilog.args` in the `snap.nc` directory. This file contains all of the command-line options that were used when you invoked *ncverilog*.
5. Parses the command line. *ncverilog* uses the Verilog-XL command-line parser to determine the search order of your directory structure.
6. Creates library directories for any libraries specified with `-y` or `-v` options.
7. Creates a `cds.lib` and `hdl.var` file in the `INCA_libs` directory.
8. Maps the Verilog-XL command-line options to create separate argument vectors for *ncvlog*, *ncelab*, and *ncsim*.
9. Invokes the three tools sequentially. If any tool fails, the next tool is not invoked. All tools share a common log file named `ncverilog.log`.

Design units contained in design files (those files specified directly on the command line) are compiled into the work library, which defaults to `worklib`.

Design units in library files (files brought in via a `-y` or `-v` option or with the `'uselib` compiler directive) are compiled into a library with the same name. For example, the following command specifies that `top.v`, `models/and2.v`, and `models/or2.v` are to be compiled into a library called `worklib`. Design units in `./libs`, which is included via the `-y` option, will be compiled into a library called `libs`.

```
% ncverilog top.v -y ./libs models/and2.v models/or2.v +libext+.v
```

**Note:** If you write your own `hdl.var` file and define the `LIB_MAP` variable to control the libraries that design units are compiled into, *ncverilog* ignores the variable. Design units in files specified directly on the command line are compiled into the work library, and design units specified in `-y` libraries or `-v` library files are compiled into libraries that have the same names.

## NC-Verilog Simulator Help

### Running NC-Verilog with the ncverilog Command

---

When *ncvlog* completes successfully, it creates a `cds.lib` and an `hdl.var` file in the `snap.nc` directory. These files contain include statements for the `cds.lib` and `hdl.var` files created during parsing. These files are passed to *ncelab* and *ncsim*.

10. Writes the `SNAPSHOT` variable to the `hdl.var` file in the `snap.nc` directory to store the name of the snapshot used in this run.

The next time you invoke *ncverilog*, it compares the current set of command-line options to the options stored in the `ncverilog.args` file. All of the plus options and dash options must be the same and in the same order for the options to be evaluated as equal.

**Note:** Some options, such as `+gui` and `-s`, which only affect run-time behavior, are not considered in the comparison.

If the options are not the same, a new `ncverilog.args` file is created, options are translated, and the tools are invoked.

If the command-line arguments are the same, *ncverilog*:

1. Reads in the `cds.lib` and `hdl.var` files in the `snap.nc` directory. The `SNAPSHOT` variable in the `hdl.var` file is used to determine what snapshot was created the last time this directory was used.
2. Determines if all source and intermediate objects are up-to-date.

If the snapshot is up-to-date, *ncsim* is invoked directly without first invoking *ncvlog* or *ncelab*. If only an SDF file has changed, only *ncelab* and *ncsim* are reinvoked.

If the snapshot is not up-to-date, all three tools are invoked. *ncvlog* is invoked with the `-update` option by default. Only design units that have changed are recompiled.

## ncverilog Command Syntax

Invoke *ncverilog* with the ncverilog command followed by the Verilog-XL command-line arguments and any ncverilog command options.

```
ncverilog verilog-xl_arguments [ncverilog_options]  
  [+all]  
  [+cdslib+path]  
  [+checkargs]  
  [+compile]  
  [+debug]  
  [+elaborate]  
  [+expand]  
  [+hdlvar+path]  
  [-h]  
  [+import]  
  [+mixedlang]  
  [+name+name]  
  [+ncelabargs+string]  
  [+ncelabexe+path]  
  [+ncerror+warning_code]  
  [+ncfatal+{warning_code | error_code}]  
  [+nclibdirname+directory_name]  
  [+ncls_all]  
  [+ncls_dependents]  
  [+ncls_snapshots]  
  [+ncls_source]  
  [+ncsimargs+string]  
  [+ncsimexe+path]  
  [+ncuid+ncuid_name]  
  [+ncversion]  
  [+ncvlogargs+string]  
  [+noautosdf]  
  [+noupdate]  
  [+ppe]  
  [-R]  
  [-r snapshot]  
  [+sdf_orig_dir]  
  [+work+library_name]
```

## ncverilog Command Options

The following list describes the ncverilog command-line options.

### **+all**

Display help on the ncverilog command and options. The +all option can only be used with the -h option. The following command lists all of the options that are recognized by the ncverilog command.

```
% ncverilog -h +all
```

If you do not include +all, only the most frequently used options are listed.

### **+cdslib+path**

Use the specified `cds.lib` file when running *ncvlog*, *ncelab*, and *ncsim*. The specified `cds.lib` file is passed to the three tools using the `ncvlog -cdslib`, `ncelab -cdslib`, and `ncsim -cdslib` options.

This option is useful if you have precompiled libraries referenced by an existing `cds.lib` file.

### **+checkargs**

Display a list of the arguments used on the command line that are recognized by *ncverilog*. This includes dash ( - ) options, plus ( + ) options, and source file arguments.

### **+compile**

Run *ncvlog* to compile the design, but do not invoke *ncelab* to elaborate the design or *ncsim* to simulate.

### **+debug**

Turn on read access to all objects in the design.

This option is the same as +ncaccess+r.

### **+elaborate**

Run *ncvlog* and *ncelab* to compile and elaborate the design, but do not invoke *ncsim* to simulate.

### **+expand**

Expand all vectors.

The NC-Verilog simulator ignores the Verilog-XL `-x` option because it is almost never necessary and because of its adverse performance impact. Use `+expand` to force the expansion of all vectors.

### **+hdlvar+path**

Use the specified `hdl.var` file when running *ncvlog*, *ncelab*, and *ncsim*. The specified `hdl.var` file is passed to the three tools using the `ncvlog -hdlvar`, `ncelab -hdlvar`, and `ncsim -hdlvar` options.

### **-h**

Display help on the `ncverilog` command and options. The help display lists only the most frequently used options.

Include the `+all` option to list all of the options that are recognized by the `ncverilog` command.

### **+import**

Prepare this Verilog design for import to VHDL. See “[Importing a Verilog-XL Design into VHDL](#)” on page 393 for details on this option.

### **+mixedlang**

Search the library structure for a VHDL binding for instances that correspond to VHDL import.

When you run `ncverilog` on a mixed-language design, the parser searches for Verilog bindings for all instances. If no Verilog binding is found for a particular instance, `ncverilog` generates an error. The `+mixedlang` option suppresses this error and instructs the

## NC-Verilog Simulator Help

### Running NC-Verilog with the ncverilog Command

---

elaborator to search for a VHDL binding for those instances for which no Verilog binding was found. If the elaborator finds a unique VHDL binding, that binding is used. If no binding is found, or if multiple bindings exist, *ncverilog* generates an error.

See [Chapter 10, “Mixed Verilog/VHDL Simulation.”](#) for information on mixed-language simulation.

#### **+name+name**

Use the specified name for the snapshot and for the `INCA_libs/snap.nc` directory.

If you run *ncverilog* and do not use the `+name+` option to specify a name, the default name of the snapshot is the name of the top-level module or the name of the first top-level module found when parsing the command line. The name of the subdirectory that contains invocation information is called `INCA_libs/snap.nc` by default.

For example, if the name of the top-level module is `top`, the following command generates a snapshot called, by default, `worklib.top:v`, and a directory called `INCA_libs/snap.nc`.

```
% ncverilog -f verilog.vc
```

The following command generates a snapshot called `worklib.myrun:v`, and a directory called `INCA_libs/myrun.nc`.

```
% ncverilog -f verilog.vc +name+myrun
```

The `+name+` option provides some of the features available with the `+ncuid+` option. If you use both `+name+` and `+ncuid+` on the command-line, the `+name+` option applies only to the snapshot name. For example, the following command generates a snapshot called `worklib.RTL:test1` and an invocation information directory called `INCA_libs/test1.nc`.

```
% ncverilog -f verilog.vc +ncuid+test1 +name+RTL
```

The following command generates a snapshot called `worklib.RTL:foo` and an invocation information directory called `INCA_libs/test1.nc`.

```
% ncverilog -f verilog.vc +ncuid+test1 +name+RTL:foo
```

#### **+ncelabargs+string**

Pass the specified `ncelab` command options to the elaborator before invoking it.

To specify an option that takes an argument, enclose the option and argument in quotation marks. To specify more than one option, enclose the list of options in quotation marks.

See [“ncelab Command Options”](#) on page 188 for a complete list of elaborator options.

## NC-Verilog Simulator Help

### Running NC-Verilog with the ncverilog Command

---

#### Examples:

```
% ncverilog -f verilog.vc +ncelabargs+-nostdout  
% ncverilog -f verilog.vc +ncelabargs+-errormax 10"  
% ncverilog -f verilog.vc +ncelabargs+-access +r+w -mindelays"
```

*ncelab* checks whether the options specified on the command line are compatible with the translated options. An error message is issued if there is a conflict. For example, an error message is issued if you have `+maxdelays` in an XL arguments file and use `+ncelabargs+-mindelays` on the command line.

*ncverilog* reads an existing `hdl.var` file. Options to *ncelab* can be specified in an `hdl.var` file with the `NCELABOPTS` variable. For example:

```
#hdl.var file  
DEFINE NCELABOPTS -access +rwc
```

See “[The hdl.var File](#)” on page 118 for details on the `hdl.var` file.

**Note:** Using `+ncelabargs+` to pass options directly to *ncelab* is strongly discouraged in general. In particular, do not use this option with the `+ncuid+` option.

#### **+ncelabexe+path\_to\_ncelab**

Invoke the specified elaborator when spawning *ncelab*. Use this option when an elaborator with statically linked PLI must be used.

#### **+ncerror+warning\_code**

Increase the severity level of the specified warning message from warning to error. The `warning_code` argument is the message code (mnemonic) that appears in the warning message following the severity code.

#### Example:

```
% ncverilog +ncerror+CUVWSP source.v
```

You can include multiple `+ncerror+` options on the command line.

Using this option can change the behavior of the tool because functions that return errors instead of warnings may behave differently. Warnings that are changed to errors are counted in the error count limit that you specify with the `+ncerrormax+` option.

### **+ncfatal+{warning\_code | error\_code}**

Increase the severity level of the specified warning message or error message from warning or error to fatal. The *warning\_code* or *error\_code* argument is the message code (mnemonic) that appears in the message following the severity code.

Example:

```
% ncverilog +ncfatal+DLCPTH source.v
```

You can include multiple `+ncfatal+` options on the command line.

### **+nclibdirname+directory\_name**

Store the snapshot, packed library file, and other generated objects in a directory with the specified name.

By default, *ncverilog* creates a directory called `INCA_libs` in the current working directory to store these objects. Use the `+nclibdirname+` option to specify a different directory. *ncverilog* automatically creates the directory for you.

The *directory\_name* argument can be a relative or absolute path to the directory. For example:

```
+nclibdirname+foo          // Stores objects in ./foo

+nclibdirname+./foo         // Stores objects in ./foo

+nclibdirname+foo/bar       // Stores objects in ./foo/bar. Directory foo must
                           // exist. ncverilog will create directory bar.

+nclibdirname+./foo/bar     // Stores objects in ./foo/bar

+nclibdirname+../foo        // Stores objects in ../foo

+nclibdirname+../foo/bar    // Stores objects in ../foo/bar

+nclibdirname+/_tmp         // Stores objects in /tmp

+nclibdirname+..             // Stores objects in .. (upper directory)
```

## NC-Verilog Simulator Help

### Running NC-Verilog with the ncverilog Command

---

#### **+ncls\_all**

List all of the objects in all libraries.

#### **+ncls\_dependents**

Show the dependents for each object. For example:

```
% ncverilog +ncls_dependents
ncverilog: v3.0.(d2): (c) Copyright 1995 - 1999 Cadence Design Systems, Inc.
    module libs.comb_logic:v (VST)           [ 1,883 bytes]
    module libs.comb_logic:v (SIG) <0x68be1e66> [      764 bytes]

        Dependents:
        module libs.comb_logic:v (VST)
            ...
            ...
            module worklib.top:v (COD) <0x21672894>      [ 3,386 bytes]
                Dependents:
                module worklib.top:v (VST)
                module worklib.top:v (SIG) <0x21672894>
            snapshot worklib.top:v (SSS)           [614,742 bytes]

                Dependents:
                module libs.comb_logic:v (VST)
                module libs.comb_logic:v (SIG) <0x68be1e66>
                module models.and2:v (VST)
                    ...

```

You can use this option with the `+ncls_snapshots` option to show the dependencies that the specified snapshot has to other INCA core data objects. For example, the following command displays all of the objects that were used to build the snapshot `worklib.top:v`.

```
% ncverilog +ncls_dependents +ncls_snapshots worklib.top:v
ncverilog: v3.0.(d2): (c) Copyright 1995 - 1999 Cadence Design Systems, Inc.
    snapshot worklib.top:v (SSS) [614,742 bytes]

    Dependents:
        module libs.comb_logic:v (VST)
        module libs.comb_logic:v (SIG) <0x68be1e66>
        module models.and2:v (VST)
        module models.and2:v (SIG) <0x328ef976>
        ...
        ...

```

### **+ncls\_snapshots**

List all snapshot (SSS) objects.

### **+ncls\_source**

Show the source file dependents of each object.

The source files that are listed for a snapshot include all of the source files that were used by all dependents. You can use this option to get a list of all the necessary files for a test case (with the exception of any files used during simulation, such as files used by \$readmem).

The source file information includes the line numbers used in the individual source files. This can be useful in finding cross-file inheritance dependencies. Note that if an object is out-of-date, the line numbers reflect the original positions in the source file, not the current positions.

### **+ncsimargs+string**

Pass the specified `ncsim` command options to the simulator before invoking it.

To specify an option that takes an argument, enclose the option and argument in quotation marks. To specify more than one option, enclose the list of options in quotation marks.

See “[ncsim Command Options](#)” on page 301 for a complete list of simulator options.

Examples:

```
% ncverilog -f verilog.vc +ncsimargs+-gui  
% ncverilog -f verilog.vc +ncsimargs+"-errormax 10"  
% ncverilog -f verilog.vc +ncsimargs+"-keyfile sim.key -errormax 10"
```

*ncsim* checks whether these options are compatible with the translated options.

*ncverilog* reads an existing `hdl.var` file. Options to *ncsim* can be specified in an `hdl.var` file with the `NCSIMOPTS` variable. For example:

```
# hdl.var file  
DEFINE NCSIMOPTS -gui
```

See “[The hdl.var File](#)” on page 118 for details on the `hdl.var` file.

**Note:** Using `+ncsimargs+` to pass options to directly to *ncsim* is strongly discouraged in general. In particular, do not use this option with the `+ncuid+` option.

### **`+ncsimexe+path_to_ncsim`**

Invoke the specified simulator when spawning *ncsim*. Use this option when a simulator with statically linked PLI must be used.

### **`+ncuid+ncuid_name`**

Use the specified unique ID name to identify the current run.

The `+ncuid+` option lets you run, either sequentially or in parallel, multiple simulations using the same intermediate objects and using the same storage location so that you can save disk space as well as compilation and elaboration time.

For example, during regression testing, in which there are typically many testbench modules that all instantiate the same design, you can assign a unique name to each run using the `+ncuid+` option and then run these jobs in parallel or in sequence. Each run reuses existing intermediate objects if these objects are the objects that the process needs. New objects are generated if they are required but do not exist. If an object exists, but is not the one that the process requires, *ncverilog* automatically detects that the new object will overwrite pre-existing data and automatically renames the object using the *ncuid\_name* so that no existing data is overwritten.

The *ncuid\_name* argument must consist of only case-sensitive alphanumeric characters and the underscore character.

See “[Using +ncuid+ to Run in Regression Mode](#)” on page 79 for more information on using the `+ncuid+` option.

### **`+ncversion`**

Displays the version number of *ncverilog*.

### **`+ncvlogargs+string`**

Pass the specified *ncvlog* command options to the parser before invoking it.

To specify an option that takes an argument, enclose the option and argument in quotation marks. To specify more than one option, enclose the list of options in quotation marks.

See “[ncvlog Command Options](#)” on page 141 for a complete list of compiler options.

## NC-Verilog Simulator Help

### Running NC-Verilog with the ncverilog Command

---

#### Examples:

```
% ncverilog -f verilog.vc +ncvlogargs+-linedebug  
% ncverilog -f verilog.vc +ncvlogargs+-view myview"  
% ncverilog -f verilog.vc +ncvlogargs+-linedebug -nostdout"
```

*ncvlog* checks whether these options are compatible with the translated options.

*ncverilog* reads an existing `hdl.var` file. Options to *ncvlog* can be specified in an `hdl.var` file with the `NCVLOGOPTS` variable. For example:

```
# hdl.var file  
DEFINE NCVLOGOPTS -linedebug
```

See “[The hdl.var File](#)” on page 118 for details on the `hdl.var` file.

**Note:** Using `+ncvlogargs+` to pass options to directly to *ncvlog* is strongly discouraged in general. In particular, do not use this option with the `+ncuid+` option.

#### **+noautosdf**

Do not perform automatic SDF annotation.

The elaborator recognizes `$sdf_annotate` system tasks in the design source files, and if the `$sdf_annotate` system tasks are scheduled to run at time 0 and meet other requirements, annotation is performed automatically. Use the `+noautosdf` option if you do not want to annotate the design.

See [Chapter 17, “SDF Timing Annotation,”](#) for details on SDF annotation.

#### **+noupdate**

Do not run the parser with the default `-update` option, which prevents the writing of intermediate objects for design units that are up-to-date.

Using `+noupdate` forces intermediate file generation for all design units and could have significant negative performance impact on compilation. Use this option only when a library is potentially corrupted and a clean rebuild is necessary.

#### **+ppe**

Invoke the Post Processing Environment (PPE). See “[The SimVision Post Processing Environment](#)” in the *SimVision Analysis Environment User Guide* for details on the PPE.

**-R**

Invoke the simulator (*ncsim*) to simulate the snapshot in the `INCA_libs/worklib` directory. No source file checking of any kind is performed. The snapshot is simply loaded and simulated.

You can use this option to simulate a snapshot multiple times using different simulator command-line options. See “[Using -R and -r to Simulate a Snapshot](#)” on page 81 for details on this option.

**-r *snapshot***

Load the specified snapshot. You can use this option to restart a simulation with a snapshot saved with the Tcl `save` command. See “[Using -R and -r to Simulate a Snapshot](#)” on page 81 for details on this option.

**+sdf\_orig\_dir**

Put the compiled SDF file in same location as the original SDF file.

**+work+*library\_name***

Use the specified library as the work library.

When you run *ncverilog*, design files (that is, files specified directly on the command line) are always compiled into the work library, which defaults to `worklib`. Use the `+work+` option to specify a different library. For example, the following command specifies that `top.v`, `models/and2.v`, and `models/or2.v` are to be compiled into a library called `models`. Design units in `./libs`, which is included via the `-y` option, will be compiled into a library called `libs`.

```
% ncverilog +work+models top.v -y ./libs models/and2.v models/or2.v +libext+.v
```

You cannot use more than one `+work+` option on the command line.

## Plus Options for NC-Verilog Tools

The following table lists *ncvlog*, *ncelab*, and *ncsim* command-line options for which corresponding plus options have been provided for the *ncverilog* command. You can use these options directly on the command line instead of including the option with the `+ncvlogargs+`, `+ncelabargs+`, or `+ncsimargs+` options. For example, the following two commands are identical:

```
% ncverilog -f verilog.vc +nclinedebug  
% ncverilog -f verilog.vc +ncvlogargs+-linedebug
```

Some options, such as `+ncstatus`, are passed to all three tools.

---

<b>ncverilog Command Option</b>	<b>Translated to...</b>
<b>Parser Options</b>	
<code>+ncchecktasks</code>	<code>ncvlog <u>-checktasks</u></code>
<code>+ncieee1364</code>	<code>ncvlog <u>-ieee1364</u></code>
<code>+nclinedebug</code>	<code>ncvlog <u>-linedebug</u></code>
<code>+ncnoline</code>	<code>ncvlog <u>-noline</u></code>
<code>+ncnomempack</code>	<code>ncvlog <u>-nomempack</u></code>
<code>+ncnopragmawarn</code>	<code>ncvlog <u>-nopragmawarn</u></code>
<code>+ncpragma</code>	<code>ncvlog <u>-pragma</u></code>
<b>Elaborator Options</b>	
<code>+ncaccess[+   -]access_spec</code>	<code>ncelab <u>-access</u></code>
<code>+ncafile+access_file</code>	<code>ncelab <u>-afile</u></code>
<code>+ncanno_simtime</code>	<code>ncelab <u>-anno simtime</u></code>
<code>+nccoverage</code>	<code>ncelab <u>-coverage</u></code>
<code>+ncdisable_enht</code>	<code>ncelab <u>-disable enht</u></code>
<code>+ncepulse_neg</code>	<code>ncelab <u>-epulse neg</u></code>
<code>+ncepulse_noneg</code>	<code>ncelab <u>-epulse noneq</u></code>
<code>+ncepulse_onedetect</code>	<code>ncelab <u>-epulse onedetect</u></code>
<code>+ncepulse_oneevent</code>	<code>ncelab <u>-epulse onevent</u></code>

---

## NC-Verilog Simulator Help

### Running NC-Verilog with the ncverilog Command

---

ncverilog Command Option	Translated to...
+ncexpand	ncelab <u>-expand</u>
+ncextend_tcheck_data_limit/ <i>percent_relaxation</i>	ncelab <u>-extend tcheck data limit</u>
+ncextend_tcheck_reference_limit/ <i>percent_relaxation</i>	ncelab <u>-extend tcheck reference limit</u>
+ncgenafile+access_filename	ncelab <u>-genafile</u>
+ncgeneric+arg	ncelab <u>-generic</u> (VHDL only)
+ncintermod_path	ncelab <u>-intermod path</u>
+ncloadplil=arg	ncelab <u>-loadplil</u>
+ncloadvpi=arg	ncelab <u>-loadvpi</u>
+ncmaxdelays	ncelab <u>-maxdelays</u>
+ncmindelays	ncelab <u>-mindelays</u>
+ncneg_tchk	ncelab <u>-neg tchk</u>
+ncnoneg_tchk	ncelab <u>-noneg tchk</u>
+ncnonotifier	ncelab <u>-nonotifier</u>
+ncno_sdfa_header	ncelab <u>-no sdfa header</u>
+ncno_tchk_msg	ncelab <u>-no tchk msg</u>
+ncnotimingchecks	ncelab <u>-notimingchecks</u>
+ncntc_warn	ncelab <u>-ntc warn</u>
+ncomicheckinglevel+arg	ncelab <u>-omicheckinglevel</u>
+ncpathpulse	ncelab <u>-pathpulse</u>
+ncplinoptwarn	ncelab <u>-plinoptwarn</u>
+ncplinowarn	ncelab <u>-plinowarn</u>
+ncpreserve	ncelab <u>-preserve</u> (VHDL only)
+ncpulse_e/arg	ncelab <u>-pulse e</u>
+ncpulse_int_e/arg	ncelab <u>-pulse int e</u>
+ncpulse_int_r/arg	ncelab <u>-pulse int r</u>
+ncpulse_r/arg	ncelab <u>-pulse r</u>

# NC-Verilog Simulator Help

## Running NC-Verilog with the ncverilog Command

---

ncverilog Command Option	Translated to...
+ncsdf_cmd_file+arg	ncelab <u>-sdf cmd file</u>
+ncsdf_nocheck_celltype	ncelab <u>-sdf nocheck celltype</u>
+ncsdf_no_warnings	ncelab <u>-sdf no warnings</u>
+ncsdf_precision+precision	ncelab <u>-sdf precision</u>
+ncsdf_verbose	ncelab <u>-sdf verbose</u>
+ncsdf_worstcase_rounding	ncelab <u>-sdf worstcase rounding</u>
+nctimescale+arg	ncelab <u>-timescale</u>
+nctypdelays	ncelab <u>-typdelays</u>

### Simulator Options

+nbasync	ncsim <u>-nbasync</u>
+ncappend_key	ncsim <u>-append key</u>
+ncinput+arg	ncsim <u>-input</u>
+nckeyfile+arg	ncsim <u>-keyfile</u>
+nclicqueue	ncsim <u>-licqueue</u>
+ncnokey	ncsim <u>-nokey</u>
+ncnolicpromote	ncsim <u>-nolicpromote</u>
+ncprofile	ncsim <u>-profile</u>
+ncprofthread	ncsim <u>-profthread</u>
+ncredmem	ncsim <u>-redmem</u>
+ncunbuffered	ncsim <u>-unbuffered</u>
+ncvcdextend	ncsim <u>-vcdextend</u>
+ncxlstyle_units	ncsim <u>-xlstyle units</u>
+nolicsuspend	ncsim <u>-nolicsuspend</u>

### Other Options

+ncappend_log	 <u>-append_log</u> Append log information from tools to <i>ncverilog</i> log file.
---------------	--

## NC-Verilog Simulator Help

### Running NC-Verilog with the ncverilog Command

---

<b>ncverilog Command Option</b>	<b>Translated to...</b>
+nccdslib+path	-cdslib <i>cdslib_path</i> Use the specified cds.lib file.
+ncerrormax+arg	-errormax <i>integer</i>
+nchdlvar+path	-hdlvar <i>hdlvar_path</i> Use the specified hdl.var file.
+ncneverwarn	-neverwarn Do not print any warning messages.
+ncnocopyright	-nocopyright Do not display copyright banners.
+ncnolog	-nolog Do not generate a log file.
+ncnostdout	-nostdout Do not print output to the screen.
+nowarn+arg	-nowarn <i>warning_code</i> Suppress warnings with the specified warning code.  To suppress multiple warnings, use multiple +nowarn+ options. For example:
	+nowarn+TFNPC +nowarn+MEMODR
+ncstatus	-status Display runtime status.

---

## Verilog-XL Command-Line Options Translation

*ncverilog* translates all applicable Verilog-XL command-line options into options for *ncvlog*, *ncelab*, and *ncsim*. The tables in this section list the XL options that are translated.

### Verilog-XL Dash (-) Option Translation Table

The following table shows what action is taken by *ncverilog* for the Verilog-XL command-line *dash* ( - ) options.

**Table 4-1 Dash Option Translation**

Verilog-XL Option	Action
-a	Ignored.
-c	Compile and elaborate only.
-d	Ignored.
-f <i>filename</i>	Read the command-line arguments contained in the specified file.  The arguments file can contain Verilog-XL command-line options, <i>ncverilog</i> command-line options, and a list of source files.  If you generate a log file, it will contain a list of all arguments contained in the arguments file specified with -f, as well as any arguments specified on the command line.
-i <i>filename</i>	Ignored with a warning about command language difference.  The NC-Verilog simulator command language is based on Tcl. You cannot use the -i option to specify an input file containing Verilog commands. You can specify a file of Tcl commands with the +tcl+ <i>filename</i> option or with the +ncinput+ <i>filename</i> option. See <a href="#">Chapter 12, “Using the Tcl Command-Line Interface,”</a> for information on the Tcl commands.
-k <i>filename</i>	Generate a keyfile with the specified name. The default name for the keyfile is <i>ncsim.key</i> .

## NC-Verilog Simulator Help

### Running NC-Verilog with the ncverilog Command

---

**Table 4-1 Dash Option Translation**

Verilog-XL Option	Action
<code>-l filename</code>	Generate a logfile with the specified name. <i>ncverilog</i> generates one log file containing information from all tools. The default name for the logfile is <code>ncverilog.log</code> .
<code>-p filename</code>	Ignored.
<code>-q</code>	Turn off the copyright banner and do not display messages for all tools.
<code>-r filename</code>	Ignored. <i>ncverilog</i> has a <code>-r snapshot</code> option that you can use to restart a simulation on a snapshot saved with the <code>save</code> command. See “ <a href="#">Using -R and -r to Simulate a Snapshot</a> ” on page 81 for more information.
<code>-s</code>	Enter interactive mode after invoking the simulator. Translated to <code>ncsim -tcl</code> .
<code>-t</code>	Ignored.
<code>-u</code>	Convert all lowercase letters to uppercase, except for text within strings, so that a source description becomes case insensitive. Translated to <code>ncvlog -upcase</code> .
	Identifiers in the NC-Verilog simulator are case sensitive. For example, if you have a signal called <code>Sum</code> and compile without <code>-upcase</code> , you must use <code>Sum</code> when you refer to the signal. If you compile with <code>-upcase</code> , you must use <code>SUM</code> .
<code>-v filename</code>	Include the specified library file.  <b>Note:</b> The argument to <code>-v</code> must be a file name. The following example will result in an error message:  <code>-v abc</code> (where “abc” is a directory, not a file)
<code>-w</code>	Ignored.  Use <code>+nowarn+warning_code</code> to turn off selected messages or <code>+ncneverwarn</code> to turn off all warning messages.
	Examples:  <code>% ncverilog -f verilog.vc +nowarn+CUVWSP</code> <code>% ncverilog -f verilog.vc +ncneverwarn</code>
<code>-x</code>	Ignored with a warning.  Use <code>+ncelabargs+-expand</code>

**NC-Verilog Simulator Help**  
Running NC-Verilog with the ncverilog Command

---

**Table 4-1 Dash Option Translation**

Verilog-XL Option	Action
<code>-y library</code>	Include the specified library directory.  <b>Note:</b> The argument to <code>-y</code> must be a directory name. The following examples will both result in error messages: <code>-y +1</code> <code>-y -y dir</code>
All -- options	Ignored.

**Verilog-XL Plus (+) Option Translation Table**

The following table shows which Verilog-XL command-line *plus* options are translated by *ncverilog*. There are many other plus options that have no counterparts in NC-Verilog or that are ignored with a warning.

**Table 4-2 Plus Option Translation**

Verilog-XL Plus (+) Option	Action
<code>+define+arg</code>	<code>ncvlog -define arg</code>
<code>+delay_mode_distributed</code>	<code>ncelab -delay mode distributed</code>
<code>+delay_mode_none</code>	<code>ncelab -delay mode none</code>
<code>+delay_mode_path</code>	<code>ncelab -delay mode path</code>
<code>+delay_mode_unit</code>	<code>ncelab -delay mode unit</code>
<code>+delay_mode_zero</code>	<code>ncelab -delay mode zero</code>
<code>+gui</code>	<code>ncsim -gui</code>
<code>+includir+arg</code>	<code>ncvlog -includir arg</code>
<code>+libext+arg</code>	Specifies the library directory file extensions, as in Verilog-XL.
<code>+liborder</code>	Scans the library files and directories in the same order they are scanned if you use this option in Verilog-XL.

**NC-Verilog Simulator Help**  
Running NC-Verilog with the ncverilog Command

---

**Table 4-2 Plus Option Translation**

<b>Verilog-XL Plus (+) Option</b>	<b>Action</b>
+librescan	Scans the library files and directories in the same order they are scanned if you use this option in Verilog-XL.
+libverbose	ncelab <a href="#"><u>-libverbose</u></a>
+licq*	ncsim <a href="#"><u>-licqueue</u></a>
+loadpli1	ncelab <a href="#"><u>-loadpli1</u></a>
+loadvpi	ncelab and ncsim <a href="#"><u>-loadvpi</u></a>
+maxdelays	ncelab <a href="#"><u>-maxdelays</u></a>
+max_err_count+num	ncvlog, ncelab, and ncsim <a href="#"><u>-errormax integer</u></a>
+max_error_count+num	
+mindelays	ncelab <a href="#"><u>-mindelays</u></a>
+multisource_int_delays	ncelab <a href="#"><u>-intermod path</u></a>
+neg_tchk	ncelab <a href="#"><u>-neg_tchk</u></a>
+nolibcell	Suppresses the default behavior of treating library modules as cells.
+no_notifier	ncelab <a href="#"><u>-nonotifier</u></a>
+no_pulse_msg	ncsim <a href="#"><u>-epulse no msg</u></a>
+nosdfwarn	ncelab <a href="#"><u>-sdf no warnings</u></a>
+noshow_cancelled_e	ncelab <a href="#"><u>-epulse noneg</u></a>
+no_show_cancelled_e	
+notchkmsg	ncelab <a href="#"><u>-no tchk msg</u></a>
+no_tchk_msg	
+notimingchecks	ncelab <a href="#"><u>-notimingchecks</u></a>
+nowarn+arg	Maps a limited set of Verilog-XL warning messages to NC-Verilog simulator warning messages. See <a href="#"><u>"Warning Message Mapping"</u></a> on page 78 for more information.

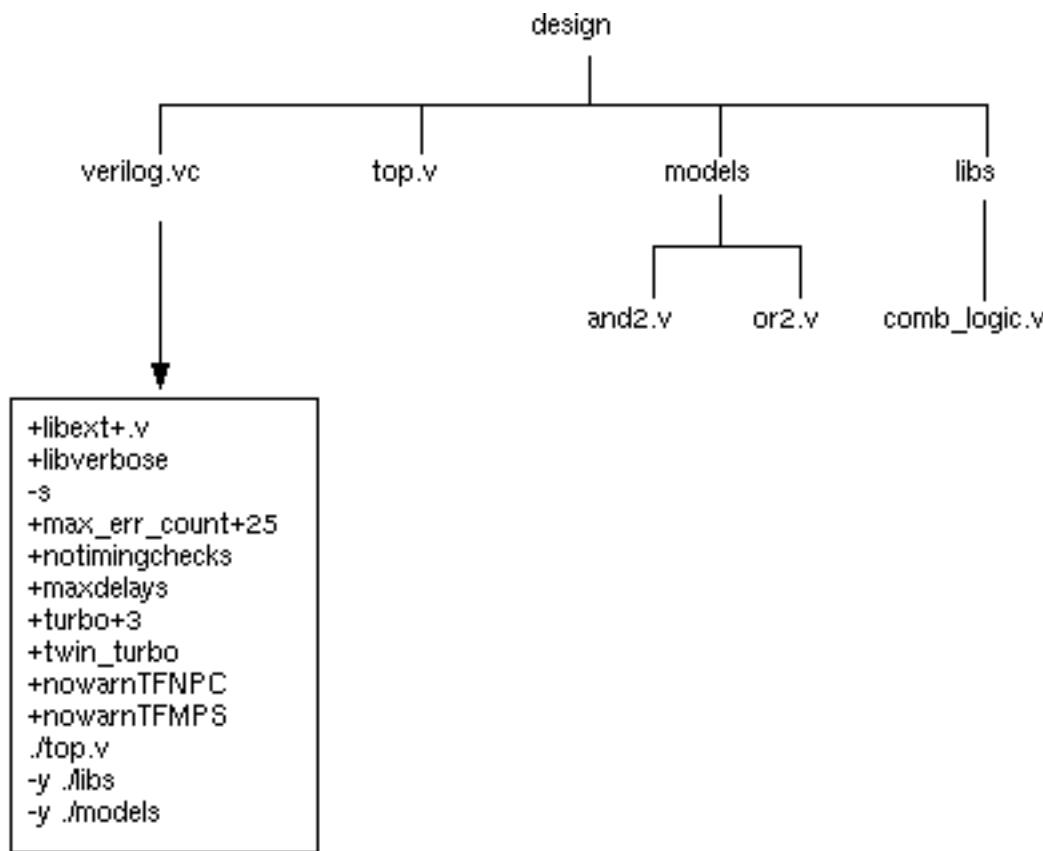
**NC-Verilog Simulator Help**  
Running NC-Verilog with the ncverilog Command

**Table 4-2 Plus Option Translation**

<b>Verilog-XL Plus (+) Option</b>	<b>Action</b>
+no_pulse_msg, +noplsmsg	ncsim <u>-epulse no msg</u>
+ntc_warn	ncelab <u>-ntc warn</u>
+pathpulse	ncelab <u>-pathpulse</u>
+pulse_e/arg	ncelab <u>-pulse e arg</u>
+pulse_r/arg	ncelab <u>-pulse r arg</u>
+pulse_int_e/arg	ncelab <u>-pulse int e arg</u>
+pulse_int_r/arg	ncelab <u>-pulse int r arg</u>
+pulse_e_style_ondetect	ncelab <u>-epulse ondetect</u>
+pulse_e_style_onevent	ncelab <u>-epulse onevent</u>
+sdf_nocheck_celltype	ncelab <u>-sdf nocheck celltype</u>
+sdf_nowarnings	ncelab <u>-sdf no warnings</u>
+sdf_no_warnings	ncelab <u>-sdf no_warnings</u>
+sdf_verbose	ncelab <u>-sdf verbose</u>
+show_cancelled_e	ncelab <u>-epulse neg</u>
+transport_int_delays	ncelab <u>-intermod path</u>
+typdelays	ncelab <u>-typdelays</u>
+vcw	ncsim <u>-gui</u>
+venv	ncsim <u>-gui</u>

## Example ncverilog Run

In the following example, source files are in the directory structure shown below. Click on [top.v](#) to see the source files used in the example.



```
% ncverilog -f verilog.vc
ncverilog: v2.2.(b2): (c) Copyright 1995 - 1999 Cadence Design Systems, Inc.
// Module top compiled into the work library, worklib.
file: ./top.v
    module worklib.top:v
        errors: 0, warnings: 0
// Module comb_logic in comb_logic.v in the -y library libs compiled into
// library libs.
file: ./libs/comb_logic.v
    module libs.comb_logic:v
        errors: 0, warnings: 0
```

## NC-Verilog Simulator Help

### Running NC-Verilog with the ncverilog Command

---

```
// Modules or2 and and2 in or2.v and and2.v in the -y library models compiled
// into library models.

file: ./models/or2.v
    module models.or2:v
        errors: 0, warnings: 0

file: ./models/and2.v
    module models.and2:v
        errors: 0, warnings: 0
        Caching library 'models' .... Done
        Caching library 'worklib' .... Done
        Caching library 'libs' .... Done

    Elaborating the design hierarchy:
    Building instance overlay tables: ..... Done
    Generating native compiled code:
        models.and2:v <0x38872c8b>
            streams: 0, words: 0
        models.or2:v <0x38872c8b>
            streams: 0, words: 0
        worklib.top:v <0x24cea274>
            streams: 4, words: 421

    Loading native compiled code: ..... Done
    Building instance specific data structures.

    Design hierarchy summary:
        Instances Unique
        Modules:      4      4
        Registers:   2      2
        Scalar wires: 4      -
        Initial blocks: 2      2
        Cont. assignments: 0      2
        Pseudo assignments: 2      2

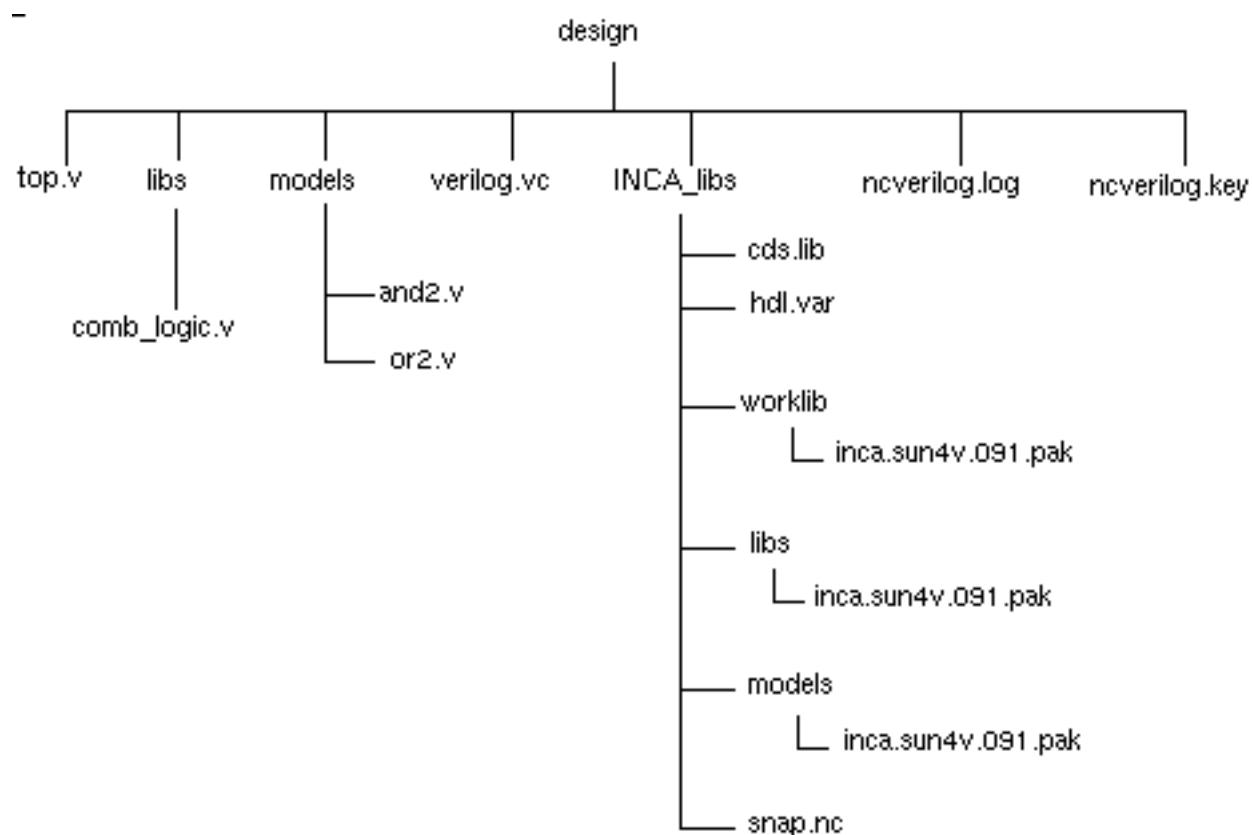
        Writing initial simulation snapshot: worklib.top:v
        Loading snapshot worklib.top:v ..... Done
ncsim>
```

## NC-Verilog Simulator Help

### Running NC-Verilog with the ncverilog Command

---

This run of *ncverilog* creates the following directory structure:



## Mapping of Warning Messages

Some Verilog-XL warning messages are commonly suppressed using the `+nowarn+` option. Because NC-Verilog simulator warning messages may have different warning codes, equivalent warning messages may be displayed.

Several of the Verilog-XL warning messages that users often suppress have been mapped to the corresponding NC-Verilog simulator messages so that using the Verilog-XL `+nowarn+` option will suppress these messages.

## NC-Verilog Simulator Help

### Running NC-Verilog with the ncverilog Command

---

The following table shows which Verilog-XL warning messages have been mapped to NC-Verilog simulator messages.

**Table 4-3 Warning Message Mapping**

Verilog-XL Warning	NC-Verilog Warning	Meaning
TFMPC	CUVWSP	Too few module port connections
TFMPS	CUVWSP	Too few module port connections
TFNPC	CUVWSP	Too few module port connections
IWFA	CSINFI	Implicit wire has no fanin
TMREN	MACRDF	Text macro redefined
TMOVR	MACNDF	Command line macro not overridden
PCDPC	CUVMPW	Port sizes differ in port connection
WLBN	CSAOBW	wand/wor connected to bidir

You can use the `+nowarn+warning_code` option to suppress warning messages from NC-Verilog simulator tools. This option is passed to all tools to suppress the message with the specified warning code.

## Updating Design Changes

If you make a change to a design source file and then rerun *ncverilog*, the parser is run with the default `-update` option. Only design units that have changed are recompiled. The design is then automatically re-elaborated and simulated.

If you make a change to any SDF-related file (the SDF source file, the compiled SDF file, or the SDF configuration file), and then rerun *ncverilog*, the elaborator automatically re-annotates the design using the new, up-to-date files. See “[SDF Annotation](#)” on page 85 for more information on SDF annotation.

If you run the NC-Verilog simulator using the *ncverilog* command, the snapshot is marked internally as a snapshot that was generated by *ncverilog*. You cannot invoke *ncupdate* or

execute the `ncsim -update` command on a snapshot of this type. Rerun *ncverilog* to update the snapshot.

## Using `+ncuid+` to Run in Regression Mode

The `ncverilog +ncuid+ncuid_name` option enables functionality in *ncverilog* that lets you run multiple simulations using the same intermediate objects and the same storage locations.

During regression testing, in which there are typically many testbench modules that instantiate the same design, you can save disk space as well as compilation and elaboration time by using the same `INCA_libs` storage location and by reusing intermediate objects. The `+ncuid+` option enables this functionality by providing a unique ID name for each simulation.

For example, suppose that you have three testbenches that test the same design. You can invoke the simulations with the following commands:

```
% ncverilog tbench1.v -y ./libs -y ./models +libext+.v +ncuid+test1  
% ncverilog tbench2.v -y ./libs -y ./models +libext+.v +ncuid+test2  
% ncverilog tbench3.v -y ./libs -y ./models +libext+.v +ncuid+test3
```

When you run *ncverilog* (in parallel or in sequence), each run reuses existing intermediate objects if these objects are the objects that the process needs. New objects are generated if they are required but do not exist. If an object exists, but is not the one that the current process requires, *ncverilog* automatically detects that the new object will overwrite pre-existing data and renames that object using the `ncuid_name` so that the new object does not overwrite existing data.

This functionality only affects data that is both written and read by the tools for the purpose of compilation, elaboration, or simulation. This data includes:

- The contents of the library system.
- The contents of the `INCA_libs` directory, including the `INCA_libs/snap.nc` invocation information.
- The compiled SDF file.

**Note:** Output data, such as log files and waveform databases, are not affected by the `+ncuid+` option. *ncverilog* does not detect that these files will be overwritten, and does not rename these files. You must use mechanisms, such as `-l logfile`, `$test$plusarg()`, and `mc_scan_plusargs()`, to rename these files to ensure that output files are uniquely identified to avoid any data collision from multiple invocations.

## NC-Verilog Simulator Help

### Running NC-Verilog with the ncverilog Command

---

The `+ncuid+` functionality creates new objects and renames them if necessary to avoid overwriting existing data. *ncverilog* renames different objects in different ways. For example:

- The `INCA_libs/snap.nc` invocation information directory is always renamed to `INCA_libs/ncuid_name.nc` because each invocation must have a unique invocation directory. For example, the following command generates `INCA_libs/test1.nc`:  

```
% ncverilog tbench1.v +ncuid+test1
```
- Snapshots are always named `lib.cell:ncuid_name` because *ncverilog* must generate a unique snapshot for each unique user-supplied name. For example, the following command generates a snapshot called `worklib.top:test1` (assuming that the top-level module is called `top`):  

```
% ncverilog tbench1.v +ncuid+test1
```
- *ncverilog*'s automatic SDF annotation requires a compiled SDF file. The compiled SDF file generated by the first run is called `sdf_filename.x`. In subsequent runs, *ncverilog* uses this file if it is the file that the current process needs. If the current process requires a different SDF file, that SDF file is compiled and is renamed `sdf_filename.ncuid_name.X` (for example, `dcache.sdf.test1.X`).

You can use the `-R` option to rerun a simulation with a unique ID. For example, suppose that you have run two simulations on the same design using different testbenches, as follows:

```
% ncverilog source.v +ncuid+test1  
% ncverilog source.v +ncuid+test2
```

*ncverilog* has generated two snapshots in the `INCA_libs/worklib` directory: `worklib.top:test1` and `worklib.top:test2`. You can rerun the second simulation using the following command:

```
% ncverilog -R +ncuid+test2 [different_simulator_options]
```

See “[Using -R and -r to Simulate a Snapshot](#)” on page 81 for details on using the `-R` option.

## Using -R and -r to Simulate a Snapshot

*ncverilog* has two command-line options that you can use to start a simulation directly if you already have a snapshot: **-R** and **-r**. If you use either of these options, *ncverilog* does not perform any kind of source file checking. The snapshot is simply loaded and simulated.

### Using -R to Simulate a Snapshot Multiple Times

The **-R** option lets you simulate the same snapshot multiple times using different simulator command-line options. For example, suppose that you run *ncverilog* with the following command line:

```
% ncverilog source.v +ncaccess+r -l simlog1
```

If the name of the top-level module is `top`, this run creates a snapshot in the `INCA_libs/worklib` directory called `worklib.top:v` by default. You can then run the simulation again with different simulator options by using the **-R** option. For example, the following command loads the snapshot and simulates with the `+ncreddmem` option. The simulator generates a logfile called `simlog2`.

```
% ncverilog -R +ncreddmem -l simlog2
```

You can prebuild a snapshot (by using the **-c** or `+elaborate` options) and then use the **-R** option to simulate that snapshot multiple times. For example, suppose that you use `$test$plusargs` in your Verilog code to execute different tests. You can prebuild a snapshot and then run the simulation with different user options on the command line. For example,

```
% ncverilog source.v -c  
% ncverilog -R +userarg  
% ncverilog -R +some_other_userarg
```

You can also generate snapshots in different libraries or directories and give your snapshots different names using *ncverilog* command-line options such as:

- `+work+library_name`
- `+nclibdirname+directory_name`
- `+name+snapshot_name`
- `+ncuid+ncuid_name`

These options alter the way in which *ncverilog* stores and manages data in the library. If you use any of these options for the initial compilation and elaboration, you must use the same combination of options when you use the **-R** option.

## NC-Verilog Simulator Help

### Running NC-Verilog with the ncverilog Command

---

#### **Example 1:**

In this example, you have run two simulations on the same design using different testbenches, and have used the `+ncuid+` option to give the runs unique names, as shown in the following commands:

```
% ncverilog source.v +ncuid+test1  
% ncverilog source.v +ncuid+test2
```

Two snapshots are generated in the `INCA_libs/worklib` directory: `worklib.top:test1` and `worklib.top:test2`. You can rerun the second simulation using the following command:

```
% ncverilog -R +ncuid+test2 [different_simulator_options]
```

#### **Example 2:**

In this example, you generate two snapshots using the following command lines:

```
% ncverilog source.v +elaborate  
% ncverilog source.v +elaborate +ncaccess+r +nclibdirname+MYINCA_libs +name+debug
```

The first snapshot (with no access to simulation objects) is stored in the default `INCA_libs/worklib` directory. The second snapshot is built with read access to simulation objects. This snapshot is called `worklib.debug:v`, and is stored in `MYINCA_libs/worklib`.

You can now use the `-R` option to simulate these prebuilt snapshots. To simulate the default snapshot, use the following command:

```
% ncverilog -R [simulator_options]
```

For example:

```
% ncverilog -R +ncredmem
```

To simulate the “debug” snapshot, use the following command:

```
% ncverilog -R +nclibdirname+MYINCA_libs +name+debug [simulator_options]
```

## Using -r to Simulate a Saved Snapshot

You can use the `-r` option to load a snapshot. As with the `-R` option, no source file checking is performed. The specified snapshot is simply loaded into the simulator.

One common use for the `-r` option is to load a snapshot that you have saved with the Tcl `save` command. This is analogous to using the Verilog-XL `-r` option to simulate with a save file.

The following example illustrates how to use `-r` to restart a simulation with a saved snapshot. In this example, the design is initially compiled and elaborated using the `+elaborate` option. The default snapshot is then loaded using the `-R` option. The simulation is run to time 1000, and then a snapshot called `worklib.save1:v` is saved.

```
% ncverilog -f verilog.vc +elaborate
ncverilog: v3.0.(d2): (c) Copyright 1995 - 1999 Cadence Design Systems, Inc.
file: ./top.v
      module worklib.top:v
      errors: 0, warnings: 0
file: ./libs/comb_logic.v
...
...
Elaborating the design hierarchy:
...
...
Writing initial simulation snapshot: worklib.top:v
% ncverilog -R -s
ncverilog: v3.0.(d2): (c) Copyright 1995 - 1999 Cadence Design Systems, Inc.
Loading snapshot worklib.top:v ..... Done
ncsim> run 1000
Ran until 1 US + 0
ncsim> save save1
Saved snapshot worklib.save1:v
ncsim> exit
```

To simulate the saved snapshot, specify the snapshot name with the `-r` option, as follows:

```
% ncverilog -r worklib.save1:v      // or: ncverilog -r save1
ncverilog: v3.0.(d2): (c) Copyright 1995 - 1999 Cadence Design Systems, Inc.
Loading snapshot worklib.save1:v ..... Done
ncsim> run
...
```

## NC-Verilog Simulator Help

### Running NC-Verilog with the ncverilog Command

---

If you exit the simulation and then invoke the simulator with a saved snapshot, as in the example shown above, *ncverilog* does not save your debug settings. When you exit, databases are closed, and any probes and breakpoints are deleted.

If you want to restore the full Tcl debug environment when you restart with a saved snapshot, make sure that you save the environment with the `save -environment filename` command. This command creates a Tcl script that captures the current breakpoints, databases, probes, aliases, and predefined Tcl variable values. You can then use the `+ncinput+` option when you invoke *ncverilog* to execute the script, or you can invoke *ncverilog* in interactive mode with the `-s` option and then use the `Tcl source` command to source the script.

For example:

```
% ncverilog -f verilog.vc -s
ncsim> (Open a database, set probes, set breakpoints, deposits, forces, etc.)
ncsim> run 100 ns
ncsim> save worklib.top:ckpt1
ncsim> save -environment ckpt1.tcl
ncsim> exit
% ncverilog -s -r worklib.top:ckpt1 +ncinput+ckpt1.tcl
```

If you restart with a saved snapshot in the same simulation session using the `Tcl restart` command or the *File—Simulation Checkpoint—Restart* command on the SimControl window:

- SHM databases remain open and all probes remain set.
- Breakpoints that are set at the time that you execute the restart remain set.  
**Note:** If you set a breakpoint that triggers, for example, every 10 ns (that is, at time 10, 20, 30, and so on) and restart with a snapshot saved at time 15, the breakpoint triggers at 20, 30, and so on, not at time 25, 35, and so on.
- Forces and deposits that are in effect at the time you issue a `save` command are still in effect when you restart.

You can load any snapshot with the `-r` option, including snapshots generated with the `+ncuid+` option (see ["Using +ncuid+ to Run in Regression Mode"](#) on page 79). For example, suppose that you have run two simulations on the same design using different testbenches, and have used the `+ncuid+` option to give the runs unique names, as shown in the following commands:

```
% ncverilog source.v +ncuid+test1
% ncverilog source.v +ncuid+test2
```

Two snapshots are generated in the `INCA_libs/worklib` directory: `worklib.top:test1` and `worklib.top:test2`. You can load the first snapshot with the following command:

```
% ncverilog -r worklib.top:test1 +ncuid+test1
```

**Note:** It is possible to load the snapshot `worklib.top:test1` (generated with the `+ncuid+test1` option) using the `+ncuid+test2` option, as follows:

```
% ncverilog -r worklib.top:test1 +ncuid+test2
```

This works because `ncverilog` uses the information in the `cds.lib` and `hdl.var` files stored in the `INCA_libs/test2.nc` directory to load the snapshot. While this command loads the specified snapshot, you must be aware that `ncverilog` is using invocation information that may be different from that generated by using the original `+ncuid+test1` option, which is stored in `INCA_libs/test1.nc`.

## PLI Tasks

Designs that depend on PLI or VPI routines (referred to as *user system tasks*) must be compiled and linked into a dynamic library or linked statically with the NC-Verilog simulator tools. When `ncverilog` detects a user system task, it prints a message informing you that the NC-Verilog simulator will need to be specially set up to run with your PLI or VPI code.

The PLI and VPI reference manuals show you how to link system tasks to the NC-Verilog tools. After you have created the shared libraries or the statically linked executable files, you may then need to make minor modifications to the `RUN_NC` script to use the statically linked elaborator and simulator.

**Note:** You should name the statically linked elaborator and simulator `ncsim` and `ncelab`. Then set the `PATH` system variable to find the statically linked files instead of those located in the installation directory, as shown in the following example:

```
set path = (/my_home/my_ncsim $path)
```

Use the `+ncelabexe` and `+ncsimexe` options to specify the statically linked elaborator and simulator to invoke when spawning `ncelab` and `ncsim`.

## SDF Annotation

You can annotate timing check and delay data contained in an SDF file. The NC-Verilog simulator supports SDF versions 1.0, 2.0, 2.1, and 3.0. For versions 2.0 and above, use the `SDFVERSION` statement in the header of the SDF file to specify the version.

SDF annotation is performed during elaboration. The elaborator recognizes `$sdf_annotate` system tasks in the design source files, and if the `$sdf_annotate` system tasks are scheduled to run at time 0 and meet other requirements, annotation is performed automatically.

See “[\\$sdf\\_annotate System Task](#)” on page 803 for a description of the `$sdf_annotate` system task. See “[Requirements for \\$sdf\\_annotate System Tasks](#)” on page 811 for a description of the rules that apply to the `$sdf_annotate` tasks for automatic SDF annotation.

It is possible to override the default automatic SDF annotation mechanism and force annotation by using the `+sdf_cmd_file+filename` option when you invoke *ncverilog*. See “[Using the +sdf cmd file+ Option](#)” on page 88 for more information.

The NC-Verilog simulator reads only compiled SDF files. You can specify the name of the SDF source file or the name of the compiled SDF file as an argument in a `$sdf_annotate` task.

If the elaborator determines that the `$sdf_annotate` argument is a text SDF file, it then looks for a corresponding compiled file (`sdf_filename.X`). For example, if the SDF file is `cpu.sdf`, the elaborator looks for `cpu.sdf.X`.

- If the elaborator doesn't find a corresponding compiled file, the elaborator issues a warning message and then spawns the *ncsdfc* utility to automatically compile the SDF file. The compiled SDF file is written to the directory that contains the SDF file.
- If a compiled file exists, *ncsdfc* checks to make sure that the date of the compiled file is newer than the date of the source SDF file and that the version of the compiled file matches the version of *ncsdfc*. If either check fails, the SDF file is recompiled. Otherwise, the compiled file is simply read.

You can run *ncsdfc* to compile the SDF file and then use the name of the compiled file as the argument to `$sdf_annotate`. This allows you to, for example, use an old compiled file even if a new SDF source file exists. For example, you could run *ncsdfc* to compile `cpu.sdf`, using the `-output` option to generate a compiled file called `cpu_preroute.sdf`, and then use `cpu_preroute.sdf` as the argument to `$sdf_annotate`. When you elaborate, *ncsdfc* checks that the version of the compiled file matches the version of *ncsdfc*. If it does, the file is read. If it doesn't, the `$sdf_annotate` task is ignored with a warning.

See “[ncsdfc](#)” on page 906 for details on *ncsdfc*.

## Using \$test\$plusargs to Selectively Perform Annotations

You can use an `if($test$plusargs("string"))` construct to selectively perform the annotations.

In the following example, `$test$plusargs` is used to check for the presence of plus options on the command line. If `+preroute` appears on the `ncverilog` command line, the file `preroute.sdf` is used for annotation. If `+postroute` appears on the command line, the file `postroute.sdf` is used.

```
module top;
    ...
    circuit m1(i1,i2,i3,o1,o2,o3);
    initial
        if ($test$plusargs("preroute"))
            $sdf_annotate("preroute.sdf", m1);
        else
            if ($test$plusargs("postroute"))
                $sdf_annotate("postroute.sdf", m1);
            else
                $display("No SDF annotation being done for this run");
        //stimulus and response checking
        ....
endmodule
```

## Specifying Precision

The `ncsdfc` utility always compiles the SDF file with a precision of 1 fs. The elaborator annotates each module using the precision of the module or the precision set by using the `ncelab -sdf_precision command_line` option.

## Turning Off SDF Annotation

To specify that you don't want to perform SDF annotation, you can:

- Use the `+noautosdf` option on the `ncverilog` command line.
- Comment out the `$sdf_annotate` system task(s) in the Verilog source file.

## Using the `+sdf_cmd_file` Option

If the `$sdf_annotate` system tasks in the design do not meet the requirements listed in “[Requirements for \\$sdf\\_annotate System Tasks](#)” on page 811, you can override the automatic annotation mechanism and force the annotation. To do this:

- Compile the SDF file with `ncsdfc`. For example,

```
% ncsdfc my.sdf
```

See “[ncsdfc](#)” on page 906 for details on compiling an SDF file with `ncsdfc`.

- Write an SDF command file.

See “[Writing an SDF Command File](#)” on page 791 for details on the SDF command file.

If you have run `ncverilog`, the elaborator generates warning messages if the `$sdf_annotate` system tasks in the design do not meet the requirements for automatic SDF annotation. These messages include a warning message like the following example. Use the information in the warning message to create the SDF command file.

```
ncelab: *W,CUSDFI (./test.v,12|16): To force annotation, use option  
+ncelabargs+-SDF_CMD_FILE <cmd_file_name> .
```

```
// SDF Command file  
// Cut and Paste the next set of lines into a SDF command file if it  
// is necessary to force annotation. Please remember to compile  
// test.sdf before using it.  
COMPILED_SDF_FILE = "test.sdf.X",  
SCOPE = test;
```

**Note:** At the time when the elaborator generates these warning messages, the scope argument to the `$sdf_annotate` task (the second argument) cannot be fully resolved. After cutting and pasting the line(s) in the warning message into an SDF command file, check the command statements and modify any `SCOPE` statements, if necessary, so that the proper scope is specified for annotation.

- Use the `+sdf_cmd_file+filename` option to include the SDF command file when you invoke `ncverilog`.

## Controlling SDF Annotator Output

Log and error messages generated by the elaborator and by `ncsdfc` while compiling the SDF file specified in a `$sdf_annotate()` system task are contained in `ncverilog.log`.

Detailed output from the SDF annotator is generated by default. This information is sent to the screen and to the log file specified by the log file argument of the `$sdf_annotate()`

system task. The Verilog-XL command-line options that you use to suppress warning or error messages, such as `+nosdfwarn`, `+sdf_no_warnings`, and so on, are translated to comparable NC-Verilog command-line options.

## Troubleshooting

Use the `nchelp` utility to get extended help on `ncvlog`, `ncelab`, and `ncsim` error messages. The syntax is:

```
% nchelp nc_tool error_code
```

The `error_code` can be in uppercase or in lowercase.

For example,

```
% nchelp ncsim BADOPT  
% nchelp ncsim badopt
```

If you get error messages telling you that user-defined system tasks were not found, see the PLI and VPI reference manuals for details on linking PLI and VPI code to create new `ncelab` and `ncsim` executables.

# Modeling Your Hardware

---

The NC-Verilog simulator is compliant with:

- The IEEE 1364 standard described in *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*, published by the IEEE.
- The OVI 2.0 description of the language described in the *OVII Verilog Hardware Description Language Reference Manual*, Version 2.0, published by OVI.
- The Verilog-XL implementation of the Verilog language described in the *Verilog -XL Reference Manual*.

You can use the `-ieee1364` command-line option when you compile the design with `ncvlog` and elaborate the design with `ncelab` to check your code for compatibility with the IEEE standard.

Coding style has a great impact on the performance of an event-driven simulator, such as NC-Verilog. There are usually several ways to model a specific piece of hardware, and some of these ways are more efficient than others during simulation because they create fewer events. In addition, some coding styles are more efficient than others because they allow the simulator to apply algorithms that help it to accelerate the simulation. See [Chapter 14, “Maximizing Simulation Performance,”](#) for guidelines on writing models that simulate faster.

This chapter includes the following sections:

- [Arrays of Instances](#)

This section clarifies and emphasizes some points in the standard specification and details the restrictions on arrays of instances imposed by the NC-Verilog simulator.

- [Verilog IEEE Std 1364-2001 Enhancements](#)

This section lists the enhancements to the Verilog language added in the 1364-2001 specification that are supported by NC-Verilog.

## Arrays of Instances

The ability to specify an array of instances is supported in the NC-Verilog simulator. Arrays of instances are described in the IEEE 1364 - 1995 Verilog HDL Language Reference Manual (Section 7.1). This section clarifies and emphasizes some points in the standard specification and details the restrictions imposed by the NC-Verilog simulator.

To specify an array of instances, you specify the instance name followed by a range specification, which is followed by the port or terminal list. For example, the following declaration specifies an array of four instances:

```
bufif0 ar[3:0] (out, in, en);      // array of tri-state buffers
```

### Instance Array Names

An array of instances must be named. This applies to instances of gates and UDPs, as well as to modules.

You can associate one instance identifier with only one range when you declare an array of instances. For example, the following specification is illegal because the same instance name is used for two ranges:

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

You can declare this array correctly as one array of eight instances, or as two arrays with unique names of four instances each.

```
nand t_nand[0:7] (...);
```

or:

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

### Array Range Expressions

You specify the range of an array of instances by using two constant expressions separated by a colon and enclosed within a pair of square brackets. Neither of the two constant expressions is required to be zero, and the left-hand index is not required to be larger than the right-hand index. However, an array of instances must have a continuous range.

The expressions in the range of an instance array declaration have the same restrictions as range expressions in all other declarations. They must be constant expressions with no hierarchical references.

## Port Connections

The IEEE specification includes rules for the terminal connections for an array of instances. The specification states that the bit length of each port expression in the declared instance array is to be compared with the bit length of each port or terminal in the instantiated module or primitive.

- For each port or terminal where the bit length of the instance array port expression is the same as the bit length of the port, the instance array port expression is connected to each port.
- If the bit lengths are different, each instance gets a part-select of the port expression as specified in the range, starting with the right-hand index.
- If there are too many or too few bits to connect to all the instances, an error is generated.

For example, if the bit length of a port expression in an instance array declaration is greater than the bit length of the corresponding port of the instanced module or UDP, the right-most bits of the port expression are associated with the port of the instance that corresponds to the right-most index of the instance array, and so on across to the left-most bits, which are associated with the left-most instance.

In the following example, the bit length of the port expression in the instance array declaration is 4 bits, while the bit length of the corresponding port of the instanced module is 2 bits. Each instance gets a part-select of the port expression as specified in the range. The right-most bits of the port expression ( { c , d } ) are associated with the port of `bar_array[ 0 ]`. The left-most bits ( { a , b } ) are associated with the port of `bar_array[ 1 ]`.

```
module bar(in);
    input [0:1] in;
endmodule
```

```
module top;
    reg a,b,c,d;
    bar bar_array [1:0] ( {a, b, c, d} );
endmodule
```

```
// The following version of module top results in the
// same port connections.
```

```
module top;
    reg a,b,c,d;
    bar bar_array0 ( {c,d} );
    bar bar_array1 ( {a,b} );
endmodule
```

## Ports of Instance Arrays

The port or terminal list describes how the gate, module, or UDP connects to the rest of the model. The list is enclosed in parentheses and the ports are separated by commas.

### Complex Expressions on Ports of Instance Arrays

Unlike Verilog-XL, the NC-Verilog simulator allows expressions that involve mathematical or logical operators (for example,  $a+b$ ) in port expressions on instance arrays. Every expression has a size, whether explicit or inferred. The bits of the expression are associated with instance array ports as described in “[Port Connections](#)” on page 92.

### Constants on Ports of Instance Arrays

As in complex expressions, the bits of constant expressions are associated with the corresponding ports of each array instance, as described in “[Port Connections](#)” on page 92.

As in Verilog-XL, explicit bit size specifications make a big difference in determining the association. For example, consider the following instances of a module that has one input port whose size is two bits:

```
twobit u1 [1:4] (1);
twobit u2 [1:4] (8'b1);
twobit u3 [1:4] (2'b1);
```

The port expression on the first instance, `u1`, is an error. An integer is a 32-bit expression, and there are only eight bits of instance array ports to associate them with.

The second instance is legal. The eight bits of the port expression are divided amongst the four two-bit ports so that the ports of instances 1 through 3 get `2'b00` and the port of instance 4 gets `2'b01`.

The third instance is also legal. The port expression size matches the size of each port, so `2'b01` is associated with all four of them.

## Differing Instances in an Array

The IEEE standard implies that instances in an instance array differ from each other only in that their ports are connected to different nets. This is not a restriction in the NC-Verilog simulator (or in Verilog-XL). You can set the values of parameters individually to different values using `defparam` statements, and this can result in different sizes for objects in different instances within an instance array. The only restriction is that the sizes of the ports cannot differ (see “[Port Sizes of Each Instance Must Be Equal](#)” on page 97).

The elaborator generates an error when the port sizes of instance array elements differ.

## Hierarchical References

References to instance arrays (in a hierarchical name, for example) must include an index. It is not legal to refer to the entire array or to a sub-range of it. For example, the following declaration of `ar` declares four instances that can be referenced by `ar[3]`, `ar[2]`, `ar[1]`, and `ar[0]`.

```
bufif0 ar[3:0] (out, in, en);      // array of tri-state buffers
```

## **NC-Verilog Restrictions to the IEEE Standard**

This section lists the restrictions on using arrays of instances imposed by the NC-Verilog simulator in the current release.

### **Indices in References Must be Integers**

When you reference an individual instance of an array of instances within the Verilog source code, the instance index must be an integer or an expression that can be reduced to an integer by the parser. You cannot use declared names (registers, wires, parameters, and so on).

**Note:** Verilog-XL allows any constant expression, including parameter references, but disallows non-constant expressions.

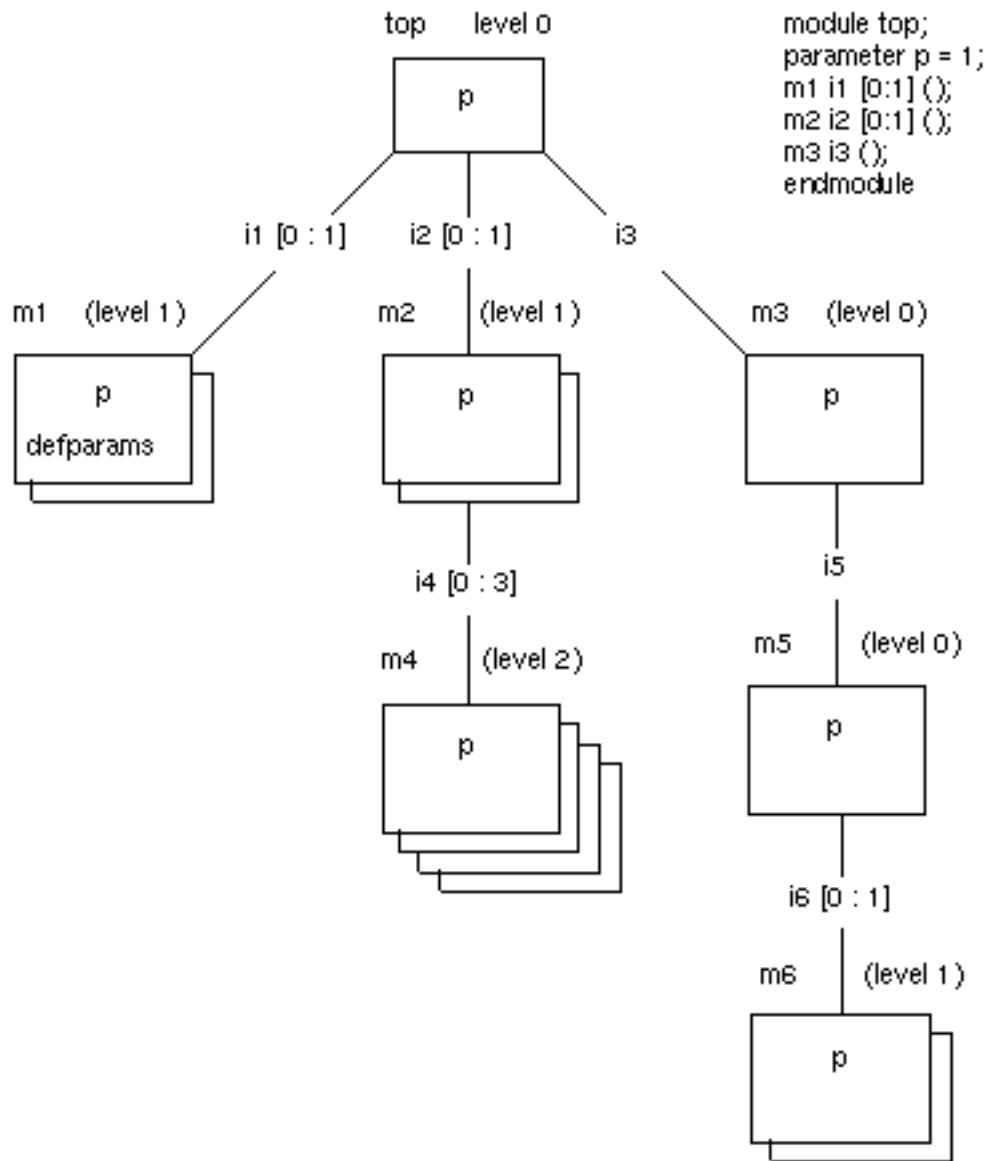
### **defparam Statements Cannot Alter the Value of Parameters in Some Modules**

A `defparam` statement cannot alter the value of a parameter in a module that is at a lesser *instance array level* than the module that contains the `defparam` statement. The instance array level of a module is the number of instance arrays in the full hierarchical name of the module instance.

The following figure clarifies what is meant by a lesser or greater instance array level.

## NC-Verilog Simulator Help

### Modeling Your Hardware



In this figure:

- Module `top`, at instance array level 0, instantiates modules `m1`, `m2`, and `m3`. Modules `m1` and `m2` are declared as arrays of instances and are at instance array level 1.
- Module `m2` instantiates module `m4`, which is declared as an array of instances. Module `m4` is at instance array level 2.

- Module `m3`, at instance array level 0, instantiates module `m5` (instance array level 0), which instantiates module `m6`, which is declared as an array of instances. Module `m6` is at instance array level 1.
- All modules contain parameter declarations in order to illustrate which `defparam` statements in module `m1` are legal and which are not legal.

Module `m1` contains the following `defparam` statements:

```
// Module m1 is at instance array level 1
module m1;
    parameter p = 2;
    defparam top.p = 3;      // Error. Trying to change a parameter in module top,
                           // which is at a lesser instance array level.

    defparam top.i1[0].p = 3;      // Legal. Changing a parameter in this module.

    defparam top.i2[0].p = 4;      // Legal. Changing a parameter in a module
                           // at the same instance array level.

    defparam top.i2[0].i4[0].p = 5; // Legal. Changing a parameter in a module
                           // at a greater instance array level.

    defparam top.i3.i5.p = 6;      // Error. Trying to change a parameter in a module
                           // at a lesser instance array level.

endmodule
```

This restriction eliminates circular dependencies in which `defparam` statements within an instance array affect the bounds of the array, and helps to implement a well-defined method for resolving parameter values in the presence of instance arrays.

**Note:** In Verilog-XL, a `defparam` statement is illegal only if it affects the range of an instance array that contains it.

## Port Sizes of Each Instance Must Be Equal

You can use `defparam` statements to give differing values to corresponding parameters within an instance array. However, it is an error if this causes the sizes of corresponding ports in each instance of an array to differ. For example:

```
module top;
    sub u1 [1:2] (4'b1010);
        defparam u1[1].p = 1;
        defparam u1[2].p = 3;
    endmodule

module sub (x);
    parameter p = 2;
    output [1:p] x;
endmodule
```

In this example, the two `defparam` statements would cause the port of instance `u1[1]` to be 1 bit and the port of instance `u1[2]` to be 3 bits. This difference in the port sizes of the two instances is not allowed, and the elaborator generates an error.

**Note:** Verilog-XL generates a warning if the port sizes of the instances are different. Verilog-XL divides the port expression bits into equal chunks and associates them with the corresponding ports without regard to the differing sizes of each port.

## Arrays of Instances and Tcl Commands

Path names in Tcl commands, like hierarchical references in the Verilog source code, can include an index expression after instances that are arrays. The index can be enclosed in square brackets or in parentheses. An instance array name with an index can be used anywhere that a normal instance name can be used. For example:

```
ncsim> value top.foo_array[0].r
ncsim> deposit foo_array[1].in 1
ncsim> describe foo_array[2].out
```

A path element that is an instance array must include an index unless it is the last path element in the name. In this case, the name refers to the entire array. Such a reference is legal only in the `describe` and `probe` commands.

The following command displays information about the instance array `foo_array`:

```
ncsim> describe foo_array
```

The following command probes all signals in all instances in the array `foo_array`:

```
ncsim> probe top.ul.foo_array
```

## Verilog IEEE Std 1364-2001 Enhancements

Many major enhancements were added to the Verilog HDL in the IEEE Std 1364-2001 specification. This section lists the enhancements that are supported in NC-Verilog.

### Comma-Separated Sensitivity List

The 1364-1995 standard specified that the event `or` operator must be used to separate signals listed in a sensitivity list. For example,

```
@(trig or enable) rega = regb;
```

The 1364-2001 standard specifies that signals in a sensitivity list can also be separated with a comma. For example,

```
@(trig, enable) rega = regb;
```

### File I/O Enhancements

The 1364-1995 standard specifies a limited set of built-in file I/O system tasks and functions. A maximum of 31 files can be opened for writing, and only ASCII characters can be written to files. The 1364-2001 standard extends the `$fopen` system task so that, in addition to opening files with multi-channel descriptors, you can open a much larger number of files for read, write, or append by using file descriptors. The file output system tasks have been extended so that the first argument to these tasks can be either a multi-channel descriptor or a file descriptor. The 1364-2001 standard also adds several new built-in file I/O tasks.

### Changes to System Tasks and Functions in the 1364-1995 Standard

This section summarizes the extensions to system tasks and functions specified in the 1364-1995 standard.

#### Tasks for Opening and Closing Files

The system tasks for opening and closing files have been extended. The `$fopen` task has been extended so that the task can now open a file with a specified name, and return either a 32-bit multi-channel descriptor or a 32-bit file descriptor, based on the absence or presence of a `type` argument. The syntax for `$fopen` is as follows:

```
integer multi_channel_descriptor = $fopen ("filename");  
| integer file_descriptor = $fopen ("filename", type);
```

The `type` argument is a character string, or a `reg` that contains a character string, which indicates how the file is to be opened. If the `type` argument is present, the file is opened as

specified by the value of `type`, and a file descriptor is returned. The `type` argument can be one of the values shown in the following table, where the “b” form distinguishes a binary file from a text file:

Argument	Description
“r” or “rb”	Open for reading
“w” or “wb”	Truncate to zero length or create for writing
“a” or “ab”	Append (open for writing at end of file)

**Note:** The 1364-2001 standard also specifies values for the `type` argument that let you open files for update (`r+`, `w+`, `a+`). NC-Verilog does not currently support opening files for update.

Example:

```
module test;
    integer fptr;
    ...

initial
begin
    fptr = $fopen("datafile", "r");
    ...
    ...
    $display( "Opened file datafile for reading" );
end
...
...
endmodule
```

File descriptors, unlike multi-channel descriptors, cannot be combined using a bit-wise `or` operation to direct output to multiple files.

The `$fclose` task closes the file specified by the multi-channel descriptor or file descriptor argument. For example,

```
$fclose(fptr);
```

## File Output System Tasks

The file output system tasks (`$fdisplay`, `$fwrite`, `$fstrobe`, `$fmonitor`, and their variants) have also been extended so that the first argument to these tasks can be either a multi-channel descriptor or a file descriptor. The new syntax is as follows:

```
file_output_task_name (multi_channel_descriptor, list_of_arguments);  
| file_output_task_name (file_descriptor, list_of_arguments);
```

**Examples:**

```
module test;  
    reg [255:0] v;  
    integer fptr;  
  
    ...  
  
    initial  
        begin  
            fptr = $fopen("output_data", "w");  
            ...  
            ...  
            $fwrite(fptr, "Value of v is: %b", v);  
            $fwrite(fptr, "Some string.");  
        end  
  
    endmodule
```

## New Built-In File I/O System Tasks

Several new system tasks have been added to the Verilog HDL.

- Formatting data to a string

There are two new string output system tasks: `$swrite` and `$sformat`.

The `$swrite` tasks (`$swrite`, `$swriteb`, `$swriteh`, `$swriteo`) are similar to the `$fwrite` family of tasks. They accept the same arguments, except that the first parameter is a `reg` variable to which the string is to be written, instead of a variable that specifies the file to which the string is to be written. The syntax is as follows:

```
$swrite (output_reg, list_of_arguments);
```

For example:

```
$swrite(output_reg, rega);  
$swriteh(output_reg, mem[0], mem[5], mem[11], mem[15]);
```

The `$sformat` task is similar to `$swrite`, except that it always interprets its second argument as a format string. The syntax is as follows:

```
$sformat (output_reg, format_string, list_of_arguments);
```

This task supports all of the format specifiers that are supported by `$display`.

For example:

```
$sformat (output_reg, "%b", rega);
$sformat (output_reg, " data = %h %o ", mem[15], mem[5], mem[11], mem[0]);
```

■ **Reading data from a file**

There are several new system tasks that let you read data from a file that was opened using a file descriptor of type `r`.

□ **Reading a character at a time**

The `$fgetc` task reads a byte from the file specified by the file descriptor. The syntax is as follows:

```
c = $fgetc (fd);
```

For example:

```
module top();
    reg [8:0] output_reg;
    integer fptr;
    ...

    initial
        begin
            fptr = $fopen ("Mem1r.inp", "r");
            ....
            ....
            output_reg = $fgetc (fptr);
        end

    endmodule
```

The `$ungetc` task inserts a specified character into a buffer specified by the file descriptor. The character will be returned by the next `$fgetc` call on that file descriptor. The syntax is:

```
code = $ungetc (c, fd);
```

For example:

```
rega = $ungetc ("A", fptr);
```

□ **Reading a line at a time**

The `$fgets` task reads characters from the file specified by the file descriptor into a `reg` variable until the variable is filled, a newline character is read and transferred to the variable, or until an end-of-file condition is encountered. The syntax is:

```
integer code = $fgets (str, fd);
```

For example:

```
module top();
    reg [3:0] mem [0:15];
    reg [858:0] output_reg;
    integer fptr, retno;
    ...

    initial
        begin
            fptr = $fopen ("Memr.inp", "r");
            retno = $fgets (output_reg, fptr);
            ...
            ...
        end
endmodule
```

□ **Reading formatted data**

The `$fscanf` task reads characters from the file specified by the file descriptor, interprets them according to a format, and stores the results in its arguments.

The `$sscanf` task reads characters from a reg variable, interprets them according to a format, and stores the results in its arguments.

The syntax is as follows:

```
integer code = $fscanf (fd, format, arguments);
integer code = $sscanf (str, format, arguments);
```

For example:

```
module top();
    reg [3:0] mem [0:15];
    reg [299:0] output_reg;
    integer fptr, retno;

    initial
        begin
            fptr = $fopen ("Mem1r.inp", "r");
            retno = $fscanf (fptr, "%b %h %o", mem[0], mem[1], mem[2]);

            output_reg = "0      1 11111  a b c d e";
            retno = $sscanf (output_reg, "%b %h %o", mem[3], mem[4], mem[5]);
            ...
            ...
        end
endmodule
```

□ **Reading binary data**

The \$fread task reads binary data from the file specified by the file descriptor into a register or into a memory. The syntax is as follows:

```
integer code = $fread (reg, fd);
integer code = $fread (mem, fd);
integer code = $fread (mem, fd, start);
integer code = $fread (mem, fd, start, count);
integer code = $fread (mem, fd, , count);
```

When \$fread is being used to load data into a memory, the data is stored starting with the lowest numbered location, continuing up to the higher location. For example, for a memory declared as memUp[ 3:15 ], the first location loaded is memUp[ 3 ], followed by memUp[ 4 ], up to memUp[ 15 ]. For a memory declared as memDown[ 15:3 ], the first location loaded is memDown[ 3 ], then memDown[ 4 ], down to memDown[ 15 ].

Start is an optional argument. If present, start is used as the starting location in memory. For example, the following task will begin loading at address 4.

```
$fread (memUp, fptr, 4);
```

Count is an optional argument. If present, count is the maximum number of locations in the memory that will be loaded. For example, for a memory declared as memUp[ 3:15 ], the following task loads 4 memory locations, beginning with memUp[ 3 ].

```
$fread (memUp, fptr, , 4);
```

■ **File positioning**

The \$ftell task returns the offset from the beginning of the file of the current byte of the file specified by the file descriptor. This offset will be read or written by a subsequent operation of that file descriptor. The syntax is:

```
integer pos = $ftell (fd);
```

For example, assume that the file Mem1r.inp contains the string ABCDEF:

```
module top();
    reg [3:0] mem [0:15];
    reg [8:0] output_reg;
    integer fptr, posno;

    initial
        begin
            fptr = $fopen ("Mem1r.inp", "r");

            posno = $ftell (fptr);      // Returns 0
```

## NC-Verilog Simulator Help

### Modeling Your Hardware

---

```
    output_reg = $fgetc (fptr);      // Read first character "A" into 0
    posno = $ftell (fptr);          // Returns 1
    $display( "Output of fgetc 1-> %c\n", output_reg);
    output_reg = $fgetc(fd);        // Read second character "B" into 1
    posno = $ftell(fd);            // Returns 2
    $display("Output of fgetc 2-> %c\n",output_reg);
    ...
    ...
endmodule
```

The `$fseek` task sets the position of the next input or output operation on the file specified by the file descriptor. The syntax for `$fseek` is:

```
code = $fseek (fd, offset, operation);
```

The `offset` argument is the number of offset bytes. The `operation` argument can be 0, 1, or 2.

- 0 Set position to offset bytes from the beginning.
- 1 Set position to current location plus offset.
- 2 Set position to EOF plus offset.

For example:

```
module top();
    reg [3:0] mem [0:15];
    reg [8:0] output_reg;
    integer fptr, retno, posno;

    initial
    begin
        fptr = $fopen ("Mem1r.inp", "r");

        posno = $ftell (fptr);
        $display ("Position = %d", posno); // Displays 0

        // Offset 2 positions from beginning (i.e., C)
        retno = $fseek (fptr, 2, 0);
        output_reg = $fgetc (fptr);
        posno = $ftell (fptr);
        $display ("Position = %d", posno); // Displays 3
        $display("Output of fgetc OP1-> %c\n", output_reg); // Displays "C"
```

```
// Offset 2 positions from 3 (i.e., F)
retno = $fseek (fptr, 2, 1);
output_reg = $fgetc (fptr);
posno = $ftell (fptr);
$display ("Position = %d", posno);
$display ("Output of fgetc OP2-> %c\n", output_reg);

// Offset 4 positions from EOF (i.e., I)
retno = $fseek (fptr, -4, 2);
output_reg = $fgetc (fptr);
posno = $ftell (fptr);
$display ("Position = %d", posno);
$display ("Output of fgetc OP3-> %c\n", output_reg);

// Go to begining of the file
retno = $fseek (fptr, 0, 0);
output_reg = $fgetc (fptr);
posno = $ftell (fptr);
$display ("Position = %d", posno);
$display ("Output of fgetc OP4-> %c\n", output_reg);
end

endmodule
```

The \$rewind task is the same as \$fseek (0, 0);

## ■ Flushing Output

The \$fflush system task writes any buffered output to the file(s) specified by the multi-channel descriptor or to the file specified by the file descriptor. The syntax of \$fflush is as follows:

```
$fflush (multi_channel_descriptor);
$fflush (file_descriptor);
$fflush ();
```

If the task is invoked with no arguments, the buffered output is written to all open files.

## ■ I/O Error Status

The \$ferror task can be used to obtain more information about an error. The syntax is:

```
integer errno = $ferror (fd, str);
```

A string description of the type of error encountered by the most recent file I/O operation is written into str. The integral value of the error code is returned in errno.

---

## Setting Up Your Environment

---

This chapter contains the following sections:

- [Overview](#)
- [The Library.Cell:View Approach](#)
- [The cds.lib File](#)
- [The hdl.var File](#)
- [The setup.loc File](#)
- [Directory Structure Example](#)

## Overview

The *ncvlog* and *ncvhdl* compilers, which parse and analyze your Verilog and VHDL source files, store compiled objects (Verilog modules, macromodules, and UDPs or VHDL entities, architectures, packages, package bodies, and configurations) and other derived data in libraries that are organized according to a Library.Cell:View (L.C:V) approach. See “[The Library.Cell:View Approach](#)” on page 109.

Three configuration files help you manage your data and control the operation of the various tools and utilities:

- `cds.lib`

Defines your design libraries and associates logical library names with physical library locations. See “[The cds.lib File](#)” on page 110.

- `hdl.var`

Defines variables that affect the behavior of tools and utilities. See “[The hdl.var File](#)” on page 118.

- `setup.loc`

Specifies the search order that tools and utilities use when searching for the `cds.lib` and `hdl.var` files. See “[The setup.loc File](#)” on page 131.

For detailed information on the library infrastructure, refer to the *Cadence Application Infrastructure User Guide*.

## The Library.Cell:View Approach

Compiled objects and other derived data are stored in libraries. The library structure is organized according to a Library.Cell:View (L.C:V) approach.

### ■ Library

A collection of related cells that describe components of a single design (a *design library*) or common components used in many designs (a *reference library*).

Each library is referenced by a logical name and has a unique physical directory associated with it. You define library names and map them to physical directories in the `cds.lib` file.

The library used for your current design work is called the *working* or *work* library. You define your current work library either by defining the `WORK` variable in the `hdl.var` file or by using the `-work` command-line option.

### ■ Cell

A cell is an object with a unique name stored in a library. Each Verilog module, macromodule, or UDP, or each VHDL entity, architecture, package, package body, or configuration is a unique cell.

The internal intermediate objects necessary to represent a cell are contained in the library database file (`.pak` file) stored in the library directory.

### ■ View

A view is a version of a cell. Views can be used to delineate between representations (schematic, VHDL, Verilog), abstraction levels (behavior, RTL, postsynthesis), status (experimental, released, golden), and so on. For example, you might have one view that is the RTL representation of a particular Verilog module and another view that is the behavioral representation, or you might have two different versions of a cell - one with timing and one without timing.

The internal intermediate objects necessary to represent a view are contained in the library database file (`.pak` file) stored in the library directory.

See “[Directory Structure Example](#)” on page 133 for an example directory structure.

## The cds.lib File

The `cds.lib` file is an ASCII text file that defines which libraries are accessible and where they are located. The file contains statements that map logical library names to their physical directory paths. During initialization, all tools that need to understand library names read the `cds.lib` file and compute the logical to physical mapping.

You can create a `cds.lib` file with any text editor. The following examples show how library bindings are specified in the `cds.lib` file with the `DEFINE` statement. The logical and physical names can be the same or different.

<b>keyword</b>	<b>logical library name</b>	<b>physical location</b>
DEFINE	<code>lib_std</code>	<code>/usr1/libs/std_lib</code>
DEFINE	<code>worklib</code>	<code>../worklib</code>

You can have more than one `cds.lib` file. For example, you can have a project-wide `cds.lib` file that contains library settings specific to a project (like technology or cell libraries) and a user `cds.lib` file. Use the `INCLUDE` or `SOFTINCLUDE` statements to include a `cds.lib` file within a `cds.lib` file.

**Note:** If you are doing a pure VHDL or a mixed-language simulation, you must use the `INCLUDE` or `SOFTINCLUDE` statement in the `cds.lib` file to include the default `cds.lib` file located in:

`your_install_directory/tools/inca/files/cds.lib`

**Note:** On Windows, the path to an included `cds.lib` file must be enclosed in quotation marks because there are spaces in the name.

This `cds.lib` file contains a `SOFTINCLUDE` statement to include a file called `cdsvhdl.lib`, which defines the Synopsys IEEE libraries included in the release. If you want to use the IEEE libraries that were shipped with Version 2.1 of the NC-VHDL simulator or NC-Sim mixed language simulator instead of the Synopsys libraries, you must include the `cds.lib` file located in:

`your_install_directory/tools/inca/files/IEEE_pure/cds.lib`

All tools and utilities that require a `cds.lib` file use the same search mechanism to find the `cds.lib` file. See [“The setup.loc File”](#) on page 131 for information on this search mechanism.

Each tool that reads a `cds.lib` file also has a `-cdslib` option that you can use on the command line to specify a `cds.lib` file. This option overrides the default search mechanism.

## The Work Library

The library used for your current design work is called the *work* or *working* library. The work library is the library into which design units are compiled. Like other libraries, the directory path of the work library is defined in the `cds.lib` file.

There are several ways to specify which library is the work library. For Verilog, you can use compiler directives in the source file, the `-work` command line option, or variables defined in the `hdl.var` file. See “[Controlling the Compilation of Design Units into Library.Cell:View](#)” on page 165 for details. For VHDL, define the `WORK` variable in the `hdl.var` file or use the `-work` option on the command line.

### **cds.lib Statements**

The following list shows the statements you can use in a `cds.lib` file.

#### **DEFINE *lib\_name path***

Associates the logical library name specified with the `lib_name` argument with the physical directory path specified with the `path` argument.

You cannot specify the same directory in multiple library definitions.

Examples:

```
DEFINE ttl_lib /usr1/libraries/ttl_lib  
DEFINE ttl ./libraries/ttl
```

#### **UNDEFINE *lib\_name***

Undefines the specified library. This command is useful for removing any libraries that were defined in other files. No error is generated if `lib_name` was not previously defined.

Example:

```
UNDEFINE ttl
```

#### **INCLUDE *file***

Reads the specified file as a `cds.lib` file. Use `INCLUDE` to include the library definitions contained in the specified file. An error message is printed if `file` is not found or if recursion is detected.

**Note:** On Windows, the path to an included `cds.lib` file must be enclosed in quotation marks because there are spaces in the name.

The file to be included does not have to be named `cds.lib`.

The following example includes the `cds.lib` file in `/users/$USER`:

```
INCLUDE /users/$USER/cds.lib
```

### **SOFTINCLUDE *file***

`SOFTINCLUDE` is the same as the `INCLUDE` statement, except that no error messages are printed if the file does not exist.

The following example includes the `cds.lib` file in the `$GOLDEN` directory:

```
SOFTINCLUDE $GOLDEN/cds.lib
```

### **ASSIGN *lib attribute path***

Assigns an attribute to the library.

**Note:** `TMP` is the only attribute that is supported.

The following example defines the `iclib` library and assigns the attribute `TMP` to the library defined as `iclib`. The value of `TMP` is `./ic_tmp_lib`.

```
DEFINE iclib ./ic_lib  
ASSIGN iclib TMP ./ic_tmp_lib
```

See “[Binding One Library to Multiple Directories](#)” on page 114 for more details on `TMP` libraries.

### **UNASSIGN *lib attribute***

Removes an assigned attribute from the library.

No error is generated if the attribute has not been assigned to the library. If the library has not been defined, an error is generated.

**Note:** `TMP` is the only attribute that is supported.

Example:

```
UNASSIGN iclib TMP
```

## **cds.lib Syntax Rules**

The following rules apply to the `cds.lib` file:

- Only one statement per line is allowed.
- Blank lines are allowed.
- Use the pound sign (#) or the double hyphen ( -- ) to begin a comment. You must precede and follow the comment character with white space, a tab, or a new line.

Examples:

```
# this is a comment  
-- this is another comment.
```

- Keywords are identified as the first non-whitespace string on a line.
- Keywords and attributes are case insensitive.
- You can include symbolic variables (UNIX environment variables like \$HOME and CSH extensions such as ~ and ~user).
- Symbolic variables and library path names are in the file system domain and are case sensitive.
- You can enter absolute or relative file paths. Relative paths are relative to the location of the file in which they occur, not to the directory where the tool was invoked.
- Library names and path names reside within the file system name-space. For Verilog, unescaped library names are the same as the Verilog name; for VHDL, unescaped library names are resolved to lower-case.

You cannot directly use escaped library names in a `cds.lib` file. To use an escaped name, run the `nmp` program in the `install_directory/tools/bin` directory to see how escaped library names are mapped to file system names. Then use the mapped name in the `cds.lib` file.

The syntax for the `nmp` program is as follows. Note the trailing space.

```
% nmp mapName {Verilog | NVerilog | vhdl} Filesys '\illegal_name '
```

For example, to use the library named `Lib*`, you must use the library's escaped name format (`\Lib*`), since "\*" is an illegal character. To determine the mapped file system name for `\Lib*`, type:

```
% nmp mapName Verilog Filesys '\Lib* '
```

The `nmp` program returns `Lib#2a`.

Use the mapped name (`Lib#2a`) in the `cds.lib` file.

## Example cds.lib File

The following example contains most of the statements you can use in a `cds.lib` file. Comments begin with the pound sign (`#`). See “[cds.lib Statements](#)” on page 111 for a description of the `cds.lib` statements.

```
# Assign /usr1/libraries/ic_library to the logical library name ic_lib
DEFINE ic_lib /usr1/libraries/ic_library
# Specify a relative path to library aludesign. The path is relative to this cds.lib
# file
DEFINE aludesign ./design
# Read cds.lib from the /users/$USERS directory.
INCLUDE /users/${USER}/cds.lib
# Read cds.lib from the $CADLIBS directory.
SOFTINCLUDE ${CADLIBS}/cds.lib
# Define a temporary directory and assign the TMP attribute to it. The directory
# ./temp must exist and templib must be set to WORK in the hdl.var file in order
# to compile data into it.
DEFINE templib ./temp_lib
ASSIGN templib TMP ./temp
```

## Binding One Library to Multiple Directories

You can bind a library that you have defined in the `cds.lib` file to a temporary storage directory by using the `ASSIGN` statement to assign the `TMP` attribute to the library. This allows multiple designers to reference a shared library, but store intermediate objects generated by the compiler or by the elaborator in separate design directories. When intermediate objects are read, the tools read whatever intermediate objects they need from the original library, and, if the objects are not in the original library, from the `TMP` library.

In the following example, a library called `asic_lib` is defined as  `${PROJECT}/asic_lib`. A temporary storage directory called `work/design_lib` is created, and the `TMP` attribute is then assigned to `asic_lib` to bind this library to the temporary storage directory.

```
# Define the shared library
DEFINE asic_lib ${PROJECT}/asic_lib
# Assign a temp storage directory
ASSIGN asic_lib TMP ./work/design_lib
```

When you compile and elaborate a design that includes design units from the shared library, all new intermediate objects are stored in the `TMP` library instead of in the `asic_lib` library.

Only one directory can be bound to a master library using the `TMP` attribute.

In the `cds.lib` file, you must define the library before you reference it with the `ASSIGN` statement. If the referenced library has not been defined before the `ASSIGN` statement is processed, the statement is ignored with a warning.

Use the `UNASSIGN` statement to remove the `TMP` attribute before compiling your design units into the master library.

Many design environments include a set of shared design libraries that have had their file system permissions set to read-only so that only an authorized user can add additional design units to, or delete or move, a shared library.

When elaborating designs that include units from these read-only libraries, the elaborator may need to produce new intermediate files for a design unit that is in a read-only library. Using an explicit `TMP` library (that is, one created by assigning the `TMP` attribute to a library) could solve this problem. However, using explicit `TMP` libraries not only requires you to add extra lines to the `cds.lib` file, but also opens up the possibility that design units could be accidentally recompiled into the `TMP` library, perhaps masking the contents of the shared design library.

To solve this problem, the elaborator can automatically create implicit `TMP` libraries. If the elaborator needs to produce new intermediate files for a design unit that is in a read-only library that has no explicit `TMP` library assigned, it will automatically create a `TMP` library and write the intermediate files into this `TMP` library. If you have used the `-messages` option, the elaborator generates a message for each implicit `TMP` library that it creates.

These implicit `TMP` libraries are co-located with the design library that contains the snapshot produced by the elaborator. Each directory for an implicit `TMP` library is named `inca.library_name` (for example, `inca.std`, `inca.ieee`, `inca.userlib`).

When a snapshot is loaded, any required intermediate files are searched for in the following order:

- The original read-only shared library
- An explicit `TMP` library associated with the shared library, if this exists
- An implicit `TMP` library

## Debugging cds.lib Files

You can use the `nchelp -cdslib` command to display information about the contents of `cds.lib` files. This can help you identify errors and any incorrect settings contained within your `cds.lib` files.

Syntax:

```
% nchelp -cdslib [cds.lib_file]
```

Examples:

```
% nchelp -cdslib  
% nchelp -cdslib ~/cds.lib  
% nchelp -cdslib ~/design/cds.lib
```

The following example shows how to display information about the contents of `cds.lib` files. In the example, the `nchelp -cdslib` command displays the contents of the `cds.lib` file that would be used. In this example, the `cds.lib` file is in the current working directory.

```
% nchelp -cdslib  
nchelp: v03.40.(p002): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.  
Parsing -CDSLIB file ./cds.lib.  
  
cds.lib files:          // The cds.lib file in the working directory includes the  
                     // cds.lib file in tools/inca/files under the  
                     // installation directory. That cds.lib file includes two  
                     // other files.  
1:  ./cds.lib  
2:  /usr1/larrybird/nccoex/tools/inca/files/cds.lib  
    included on line 2 of ./cds.lib  
3:  /usr1/larrybird/nccoex/tools/inca/files/cdsvhdl.lib  
    included on line 2 of /usr1/larrybird/nccoex/tools/inca/files/cds.lib  
4:  /usr1/larrybird/nccoex/tools/inca/files/cdsvlog.lib  
    included on line 3 of /usr1/larrybird/nccoex/tools/inca/files/cds.lib
```

Libraries defined:

Defined in ./cds.lib:

Line #	Filesys	Verilog	VHDL	Path
-----	-----	-----	-----	-----
1	worklib	worklib	WORKLIB	./INCA_libs/worklib

## NC-Verilog Simulator Help

### Setting Up Your Environment

---

Defined in /usr1/larrybird/nccoex/tools/inca/files/cdsvhdl.lib:

Line #	Filesys	Verilog	VHDL	Path
1	std	std	STD	/usr1/larrybird/nccoex/tools/inca/files/STD
2	synopsys	synopsys	SYNOPSYS	/usr1/larrybird/nccoex/tools/inca/files/SYNOPSYS
3	ieee	ieee	IEEE	/usr1/larrybird/nccoex/tools/inca/files/IEEE
4	ambit	ambit	AMBIT	/usr1/larrybird/nccoex/tools/inca/files/AMBIT
5	vital_memory	vital_memory	VITAL_MEMORY	/usr1/larrybird/nccoex/tools/inca/files/VITAL_MEMORY

Here are some common error and warning messages caused by problems with the `cds.lib` file:

```
% ncvlog board.v
ncvlog: v03.40.(p002): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
ncvlog: *W,DLNOCL: Unable to find a 'cds.lib' file to load in.
ncvlog: *F,WRKBAD: logical library name WORK is bound to a bad library name
'worklib'.
```

The `DLNOCL` warning occurs when the tool could not find a `cds.lib` file using the search order specified in the `setup.loc` file.

The `WRKBAD` error occurs when the work library is defined in the `hdl.var` file (for example, `DEFINE WORK worklib`), but the `cds.lib` file does not define the corresponding library (for example, `DEFINE worklib ./worklib`).

```
% ncvlog board.v
ncvlog: v03.40.(p002): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
ncvlog: *W,DLCPTH: cds.lib Invalid path '/usr1/clinton/inca/board/worklib' on line
7 of ./cds.lib (cds.lib command ignored).
```

The `DLCPTH` warning occurs when the directory path specified in the `cds.lib` file doesn't exist or is inaccessible. For example, you may have the following line in your `cds.lib` file, but you haven't created a `./worklib` physical directory.

```
DEFINE worklib ./worklib
```

## The hdl.var File

The `hdl.var` file is an optional configuration file. This ASCII text file can contain:

- Configuration variables, which determine how your design environment is configured. These include:
  - Variables that you can use to specify the work library where the compiler stores compiled objects and other derived data. For Verilog, you can use the `LIB_MAP` or `WORK` variables. For VHDL, use the `WORK` variable.
  - For Verilog, variables (`LIB_MAP`, `VIEW_MAP`, `WORK`) that you can use to specify the libraries and views to search when the elaborator resolves instances.
- Variables that allow you to define compiler, elaborator, and simulator command-line options and arguments.
- Variables that specify the locations of support files and invocation scripts.

The `hdl.var` file, unlike the `cds.lib` file, is optional because all of the variables that can be defined in the file are optional. However, if you do not have an `hdl.var` file, the simulator tools generate a warning message telling you that an `hdl.var` file could not be found. For example, if you do not have an `hdl.var` file, and you define the work library by using the `-work` option on the command line, the parser will generate the warning message and then compile the source files.

**Note:** Some variables that you can set in the `hdl.var` file, such as the `NCVLOGOPTS`, `NCVHDLOPTS`, `NCELABOPTS`, and `NCSIMOPTS` variables described in “[hdl.var Variables](#)” on page 121, can also be set as environment variables from the operating system. Variables that are set this way can affect tool behavior, can override variables set in the `hdl.var` file and, in some cases, can interact in undesirable ways with makefiles. No messages are generated telling you that a tool’s behavior has been modified because a variable was set from the operating system.

All tools and utilities that use an `hdl.var` file use the same search mechanism to find the file. See “[The setup.loc File](#)” on page 131 for information on this search mechanism.

Each tool that reads an `hdl.var` file also has a `-hdlvar` option that you can use on the command line to specify an `hdl.var` file. This option overrides the default search mechanism.

You can have more than one `hdl.var` file. For example, you can have a project `hdl.var` file that contains variable settings used to support all your projects and you can have local `hdl.var` files located in specific design directories that contain variable settings specific to each project, such as the setting for the `WORK` variable.

## NC-Verilog Simulator Help

### Setting Up Your Environment

---

If you define the same variable in more than one file, the last variable read is used. For example, suppose that you have the following `hdl.var` file in your current working directory. The `VERILOG_SUFFIX` variable defines recognized file extensions for Verilog source files.

```
INCLUDE ~/hdl.var
DEFINE VERILOG_SUFFIX (.ver)
DEFINE WORK ./worklib
```

The `hdl.var` file in your home directory is as follows:

```
DEFINE VERILOG_SUFFIX (.vg)
```

The first line in the `hdl.var` file includes the `hdl.var` file in your home directory. This file sets the `VERILOG_SUFFIX` variable to `.vg`. The next line then sets the same variable to `.ver`. Only this suffix (`.ver`) will be recognized as a valid suffix.

Now, suppose that the `hdl.var` file was written as follows:

```
DEFINE VERILOG_SUFFIX (.ver)
INCLUDE ~/hdl.var
DEFINE WORK ./worklib
```

In this case, the `VERILOG_SUFFIX` variable is first set to `.ver`, and then redefined to be `.vg`. Only the `.vg` suffix will be recognized.

If you want both suffixes to be recognized, you could, for example, do the following:

```
# ./hdl.var
INCLUDE ~/hdl.var
DEFINE VERILOG_SUFFIX $VERILOG_SUFFIX (.ver)
DEFINE WORK worklib

# ~/hdl.var
DEFINE VERILOG_SUFFIX (.vg)
```

In this case, `VERILOG_SUFFIX` is first set to `.vg`. Then the `.ver` suffix is appended to this definition so that the compiler will recognize both suffixes.

## hdl.var Statements

The following list shows the statements you can use in an `hdl.var` file. `variable` is an alphanumeric variable name. `value` is optional; if provided, it is either scalar or a list. See [“hdl.var Variables”](#) on page 121 for a list of `hdl.var` variables.

### **DEFINE `variable` `value`**

Defines a variable and assigns a value to the variable.

The following example defines the variable `WORK` to be `worklib`.

```
DEFINE WORK worklib
```

The following example defines `VERILOG_SUFFIX` as the list `.v`, `.vg`, and `.vb`.

```
DEFINE VERILOG_SUFFIX (.v, .vg, .vb)
```

The following example defines the variable `NCVLOGOPTS`, which is used to specify command-line options for the `ncvlog` compiler.

```
DEFINE NCVLOGOPTS -messages -errormax 10 -update
```

### **UNDEFINE `variable`**

Causes `variable` to become undefined. This statement is useful for removing definitions that were defined in other files. If `variable` was not previously defined, you will not get an error message.

```
UNDEFINE NCUSE5X
```

### **INCLUDE `filename`**

Reads `filename` as an `hdl.var` file.

Use `INCLUDE` to include the variable definitions contained in the specified file. The pathname can be absolute or relative. If it is relative, it is relative to the `hdl.var` file in which it is defined.

Examples:

```
INCLUDE ~/my_hdl.var  
INCLUDE /users/${USER}/hdl.var
```

If the file is not found, a warning message is printed.

### **SOFTINCLUDE *filename***

SOFTINCLUDE is the same as the INCLUDE statement, except that no warning message is printed if the specified file cannot be found.

Examples:

```
SOFTINCLUDE ~/hdl.var  
SOFTINCLUDE ${GOLDEN}/hdl.var
```

### **hdl.var Variables**

The following list shows the variables you can use in an hdl.var file.

**Note:** The variable definitions in the hdl.var file are treated as literal strings. Do not use quotation marks in the definitions unless you explicitly want them as part of the input. For example, use:

```
DEFINE NCVLOGOPTS -define foo=16'h03
```

instead of

```
DEFINE NCVLOGOPTS -define foo="16'h03"
```

which is the same as typing:

```
% ncvlog -define foo=\"16'h03\"
```

### **LIB\_MAP**

#### **(Verilog only)**

The LIB\_MAP variable is used by both *ncvlog* and by *ncelab*.

- For *ncvlog*, LIB\_MAP maps files and directories to library names. The definition of the variable specifies that source files are to be compiled into a particular library. Use the plus sign (+) to create a default for files or directories that are not explicitly stated.

Example:

```
DEFINE LIB_MAP ( ./design => designlib, \  
                ./source/lib1/... => lib1, \  
                myfile.v => mylib, \  
                + => worklib )
```

This definition of the LIB\_MAP variable specifies that:

- Any file in the directory ./design is to be compiled into designlib.

- ❑ All files and directories below `./source/lib1` are to be compiled into `lib1`.
- ❑ The file `myfile.v` is to be compiled into `mylib`.
- ❑ Any other file is to be compiled into `worklib`.

See “[Controlling the Compilation of Design Units into Library.Cell:View](#)” on page 165 for more details.

If the `WORK` variable and the `LIB_MAP` variable are both defined, the definition of the `WORK` variable overrides the definition of the `LIB_MAP` variable. This is important to remember if you are doing a mixed-language simulation, where you might define the `LIB_MAP` variable to control where the Verilog design units get compiled and then define the `WORK` variable to define the work library for VHDL. In this case, all design units would be compiled into the library specified with the `WORK` variable. To avoid this, you can either remove the definition of the `WORK` variable and then use the `-work` command-line option when compiling VHDL, or you can remove the definition of the `LIB_MAP` variable and then use the `-work` option when compiling Verilog.

- For `ncelab`, `LIB_MAP` specifies the list of libraries to search when resolving instances. See “[How Modules and UDPs Are Resolved During Elaboration](#)” on page 233.

## NCELABOPTS

Sets elaborator command-line options. Top-level design unit name(s) can also be included.

You cannot include the `-logfile`, `-append_log`, `-cdslib`, or `-hdlvar` options in the definition of this variable.

Example:

```
DEFINE NCELABOPTS -messages -errormax 10
```

## NCHELP\_DIR

Specifies the path to the directory where the help text files are located. These files are used by `nchelp` to print more detailed information on the messages printed by other tools. See “[nchelp](#)” on page 842.

The help files are usually located in:

`install_directory/tools/inca/files/help`

Use the `NCHELP_DIR` variable if the help files are located in another directory.

```
DEFINE NCHELP_DIR path_to_help_files
```

## **NCSDFCOPTS**

Sets command-line options for *ncsdfc*. See “[ncsdfc](#)” on page 906 for details on *ncsdfc*.

Example:

```
DEFINE NCSDFCOPTS -messages -nocopyright
```

## **NCSHELLOPTS**

Sets command-line options for *ncshell*. See “[Generating a Shell with ncshell](#)” on page 385 for details on *ncshell*.

## **NCSIMOPTS**

Sets simulator command-line options. A snapshot name can also be included.

You cannot include the `-logfile`, `-append_log`, `-cdslib`, or `-hdlvar` options in the definition of this variable.

Example:

```
DEFINE NCSIMOPTS -messages -errormax 10
```

## **NCSIMRC**

Executes a command file when *ncsim* is invoked. This command file can contain commands, such as aliases, that you use with every simulation run.

Example:

```
DEFINE NCSIMRC /usr/design/rcfile
```

## **NCUPDATEOPTS**

Sets command-line options for *ncupdate*. For example, if you have compiled a new elaborator with PLI routines statically linked, `ncupdateopts -ncelab` specifies the path to the new elaborator. See “[ncupdate](#)” on page 928 for details on *ncupdate*.

Example:

```
DEFINE NCUPDATEOPTS -ncelab ./pli/my_elab
```

## **NCUSE5X**

Generates the packed library database file, but also creates the full Cadence 5.x library structure, in which the intermediate objects for each design unit are stored in their own subdirectories under the work library. It also creates three other files for each design unit: `master.tag`, `verilog.v`, and `pc.db`.

The full 5.x library structure and the additional files make it possible for tools that do not understand the default packed library structure to access the intermediate objects that are required by the tool. For example, you must compile the source files with this variable defined (or by using the `-use5x` option on the `ncvhdl` or `ncvlog` command line) if you want to use a 5.x configuration file to control the binding of instances during elaboration.

**Example:**

```
DEFINE NCUSE5X
```

## **NCVERILOGOPTS**

**(Verilog only)**

Sets command-line options for `ncverilog` and for `ncprep`. Both `ncverilog` and `ncprep` create `hdl.var` files automatically if one doesn't exist. However, if you want to set frequently-used options, or if you want to set defaults to be used by all users, you can create an `hdl.var` file before running the tools, and the tools will read this `hdl.var` file. Use the `NCVERILOGOPTS` variable to set `ncverilog` or `ncprep` command-line options.

See [Chapter 4, “Running NC-Verilog with the ncverilog Command,”](#) for details on `ncverilog`. See [“ncprep”](#) on page 867 for details on `ncprep`.

## **NCVHDLOPTS**

**(VHDL only)**

Sets command-line options for the `ncvhdl` compiler. VHDL source file names can also be included.

You cannot include the `-logfile`, `-append_log`, `-cdslib`, or `-hdlvar` options in the definition of this variable.

**Example:**

```
DEFINE NCVHDLOPTS -messages -errormax 10 -file ./proj_file
```

## **NCVLOGOPTS**

### **(Verilog only)**

Sets command-line options for the *ncvlog* compiler. Verilog source file names can also be included.

You cannot include the `-logfile`, `-append_log`, `-cdslib`, or `-hdlvar` options in the definition of this variable.

Example:

```
DEFINE NCVLOGOPTS -messages -errormax 10 -file ./proj_file
```

## **SRC\_ROOT**

Defines an ordered list of paths to search for source files when you are updating a design either by running `ncsim -update` or by rerunning `ncverilog`. The paths listed in the definition of this variable tell the NC tools where to look for source files if design units are out-of-date.

See “[SRC\\_ROOT](#)” on page 161 for more information.

Example:

```
DEFINE SRC_ROOT (~larrybird/source, $PROJECT)
```

## **VERILOG\_SUFFIX**

### **(Verilog only)**

Defines valid file extensions for Verilog source files.

Example:

```
DEFINE VERILOG_SUFFIX (.v, .vr, .vb, .vg)
```

## **VHDL\_SUFFIX**

### **(VHDL only)**

Defines valid file extensions for VHDL source files.

Example:

```
DEFINE VHDL_SUFFIX (.vhd, .vhdl)
```

## VIEW

### (Verilog only)

Sets the view name. See “[Controlling the Compilation of Design Units into Library.Cell:View](#)” on page 165 for details on this variable.

Example:

```
DEFINE VIEW behavior
```

## VIEW\_MAP

### (Verilog only)

The VIEW\_MAP variable is used by both *ncvlog* and by *ncelab*.

- For *ncvlog*, VIEW\_MAP maps files and file extensions to view names. The definition of the variable specifies that files or files with a particular extension are compiled with a specific view name. See “[Controlling the Compilation of Design Units into Library.Cell:View](#)” on page 165.
- For *ncelab*, VIEW\_MAP is used to establish the list of views to search when resolving instances. See “[How Modules and UDPs Are Resolved During Elaboration](#)” on page 233

Example:

```
DEFINE VIEW_MAP ( .v => behav, \
                  rtl => rtl, \
                  gate => gate, \
                  myfile.v => gate)
```

## VXLOPTS

### (VHDL only)

Specifies Verilog-XL command-line options.

This variable should be used only by Leapfrog with Verilog Model Import customers who want to import the same Verilog model into NC-Sim. This variable is used by the *ncxlimport* utility

to prepare the Verilog model for import. See “[Preparing a Leapfrog Verilog Model Import Design](#)” on page 399 for details on using *ncximport*.

## WORK

Defines the current work library into which HDL design units are compiled.

Example:

```
DEFINE WORK worklib
```

You can also specify the work library with the *-work* command-line option.

## hdl.var Syntax Rules

The following rules apply to the *hdl.var* file:

- Only one statement per line is allowed.
- Keywords and variable names are case insensitive.
- Variable values, file names, and path names are case sensitive.
- Begin comments with either the pound sign (#) or a double hyphen (--). The comment character must be either the first character in a line or preceded by white space.
- You can extend a statement over more than one line by using the escape character (\) as the last character of the line. For example:

```
DEFINE ALPHA (a,\  
             b,\  
             c)
```

is the same as:

```
DEFINE ALPHA (a, b, c)
```

- Left and right parentheses indicate the beginning and end of a list of values.
- Use a comma to separate values in a list.
- You can have a list containing zero elements.

```
DEFINE EMPTY_LIST ( )
```

- Any character can be escaped using the backslash (\) escape character. Characters should be escaped if the meaning of the character is its ASCII value. For example, the following line defines the variable JUNK as \dump\.

```
DEFINE JUNK \\dump\\
```

The following example defines LIST as a, b and c.

```
DEFINE LIST (a\,b,c)
```

- You can use tilde (~) in *filename* or *value* to specify:

- ~ (or \$HOME)
  - ~*user* (home of <i>user</i>)

The “~” must be the first non-whitespace character in *filename* or *value*. For example:

```
DEFINE DIR_RELATION ~/bin != ~larrybird/bin
```

expands to:

```
/usr/bin != /usr/larrybird/bin
```

- The white space preceding and following a scalar value is ignored. In the following two lines, the variable TEST has the same value (this is a test):

```
DEFINE TEST      this is a test  
DEFINE TEST this is a test
```

- You can use the dollar sign (\$) in *filename* or *value* to indicate variable substitution. The syntax can be either \$variable or \${variable}, where the left and right braces ({} ) are real characters that mark the beginning and end of the variable name. Variable substitution first searches the hdl.var definitions and, if none are found, then searches for environment variables.

The following example uses the environment variable \$SHELL to define an hdl.var variable:

```
DEFINE MY_SHELL $SHELL
```

In the following example, LIB\_MAP is defined as:

```
(./source/lib1/... => lib1)
```

Then the variable is redefined as

```
(./source/lib1/... => lib1, ./design => lib2)  
DEFINE LIB_MAP (./source/lib1/... => lib1)  
DEFINE LIB_MAP (${LIB_MAP}, ./design => lib2)
```

In the following example, ALPHA is defined as first. Then BETA is defined as first == one.

```
DEFINE ALPHA first  
DEFINE BETA ${alpha} == one
```

- When a scalar value or file name is specified as a relative path, the path is relative to the location of the hdl.var file in which it is defined.

## Example hdl.var File

In the following example `hdl.var` file, the `WORK` variable is used to define the work library into which design units are compiled. This library must be defined in the `cds.lib` file. Other variables are defined to list valid file extensions for Verilog and VHDL source files and to specify command-line options for various tools.

```
# Define the work library
DEFINE WORK worklib
# Define valid Verilog file extensions
DEFINE VERILOG_SUFFIX (.v, .vr, .vb, .vg)
# Define valid VHDL file extensions
DEFINE VHDL_SUFFIX (.vhd, .vhdl)
# Specify command-line options for the ncvhdl compiler
DEFINE NCVHDLOPTS -messages -errormax 10
# Specify command-line options for the ncvlog compiler
DEFINE NCVLOGOPTS -messages -errormax 10 -ieee1364
# Specify command-line options for the elaborator
DEFINE NCELABOPTS -messages -errormax 10 -ieee1364 -plinoptwarn
# Specify the simulation startup command file
DEFINE NCSIMRC /usr/design/simrc.cmd
```

## Debugging hdl.var Files

You can use the `nchelp -hdlvar` command to display information about the contents of `hdl.var` files. This can help you identify incorrect settings that may be contained within your `hdl.var` files.

Syntax:

```
% nchelp -hdlvar [hdl.var_file]
```

Examples:

```
% nchelp -hdlvar
% nchelp -hdlvar ~/hdl.var
```

The following example shows how to display information about the contents of an `hdl.var` file. In the example, the `nchelp -hdlvar` command displays the contents of the first `hdl.var` file found using the search order in the `setup.loc` file. In this example the `hdl.var` file is in the current working directory.

## NC-Verilog Simulator Help

### Setting Up Your Environment

---

```
% nchelp -hdlvar
nchelp: v03.40.(p002): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
Parsing -HDLVAR file ./hdl.var.
hdl.var files:
1: ./hdl.var
Variables defined:
Defined in ./hdl.var:
Line # Name Value
----- -----
5 LIB_MAP ( /net/foghorn/usr1/belanger/chip1 => chip1 , \
            /net/foghorn/usr1/belanger/libs/misc.v => misc )
1 NCVLOGOPTS -messages
2 NCELABOPTS -messages
3 NCSIMOPTS -messages
4 VERILOG_SUFFIX ( .v , .vlog )
6 VIEW_MAP ( .g => gates , .b => behav , .rtl => rtl )
7 WORK worklib
```

## The setup.loc File

The `setup.loc` file is a file that you can write to specify the directories that you want the tools and utilities to search for the `cds.lib` and `hdl.var` files.

Tools and utilities that need to read library definition files (`cds.lib`) and configuration files (`hdl.var`) search for these files in the following way:

1. Search (in order) the following directories for a `setup.loc` file:
  - The current work directory
  - `$CDS_WORKAREA` (user work area, if defined)
  - `$CDS_SEARCHDIR` (if defined)
  - `$HOME`
  - `$CDS_PROJECT` (project area, if defined)
  - `$CDS_SITE` (site-specific location, if defined)
  - your\_install\_directory/share*
2. If a `setup.loc` file is found, use the search list in the `setup.loc` file to find the setup files.
3. If a `setup.loc` file is not found, search (in order) the following directories for the setup files:
  - The current work directory
  - `$CDS_WORKAREA` (user work area, if defined)
  - `$CDS_SEARCHDIR` (if defined)
  - `$HOME`
  - `$CDS_PROJECT` (project area, if defined)
  - `$CDS_SITE` (site-specific location, if defined)
  - your\_install\_directory/share*
4. If the setup files are not found, generate an error.

To write a `setup.loc` file, you can copy the example in `install_directory/share/cdssetup/setup.loc` and edit the file to list the directories in the order that you want them to be searched. You must put the `setup.loc` file in one of the locations listed above so that the tools can find it.

## setup.loc Syntax Rules

The following rules apply to the `setup.loc` file:

- Only one entry per line is allowed.
- Use a semi-colon ( ; ), the pound sign ( # ), or a double hyphen ( -- ) to begin a comment.
- The file can include:
  - ~
  - ~user
  - `$environment_variable`
  - `${environment_variable}`

By convention, environment variables are given uppercase names. See the documentation for your implementation of UNIX for complete details on setting environment variables.

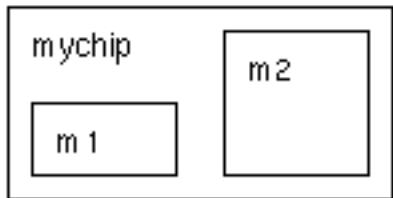
If a directory specified in `setup.loc` references an environment variable that is not set, the location referenced by the variable is not searched and no warning or error message is issued.

- Relative paths in `setup.loc` files are relative to the current directory; they are not relative to the location of the file in which they occur or to the directory where the tool was invoked.

If a directory specified in `setup.loc` cannot be found or is not accessible, the search advances to succeeding locations without printing warning or error messages.

## Directory Structure Example

In the following example, a Verilog design is used to illustrate the concepts introduced in this chapter. In the example design, a module `mychip` instantiates two other modules, `m1` and `m2`.

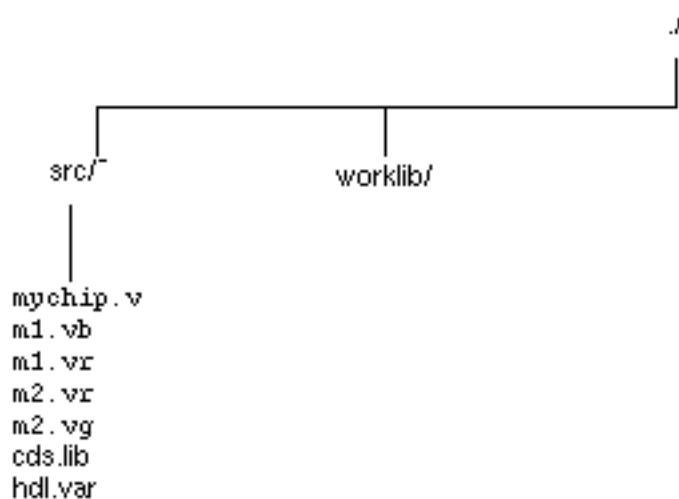


A single Verilog description of `mychip` is contained in the file `mychip.v`, but you have generated multiple descriptions of `m1` and `m2`.

- For `m1`, there is both a behavioral and an RTL description, described in `m1.vb` and `m1.vr`, respectively.
- For `m2`, there is both an RTL and a synthesized gate-level representation, described in `m2.vr` and `m2.vg`, respectively.

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

All of these source files reside in the `src/` subdirectory, from which all tools are invoked. You have created a subdirectory at the same level as `src/`, which you want to use as the work library.



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

```
# cds.lib file
DEFINE worklib ../../worklib
```

The `hdl.var` file, shown below, is also located in the `src/` directory. This file includes definitions of the `LIB_MAP` and `VIEW_MAP` variables, which can be used to specify the library and view mapping for Verilog design units.

**Note:** The `LIB_MAP` and `VIEW_MAP` variables do not apply to VHDL.

```
# Define library mapping.
# Compile all files in src/ into worklib
DEFINE LIB_MAP (./ => worklib)

# Define view mapping.
# Files with .vb extension are compiled into view beh
# Files with .vr extension are compiled into view rtl
# Files with .vg extension are compiled into view gates
# Files with .v extension are compiled into view module
DEFINE VIEW_MAP (.vb => beh, \
                 .vr => rtl, \
                 .vg => gates, \
                 .v => module)
```

## NC-Verilog Simulator Help

### Setting Up Your Environment

Use *ncvlog* to compile the design units in the source files.

```
% ncvlog mychip.v m1.vb m1.vr m2.vg m2.vr
```

When the design is compiled, a cell and a view is created for each module.

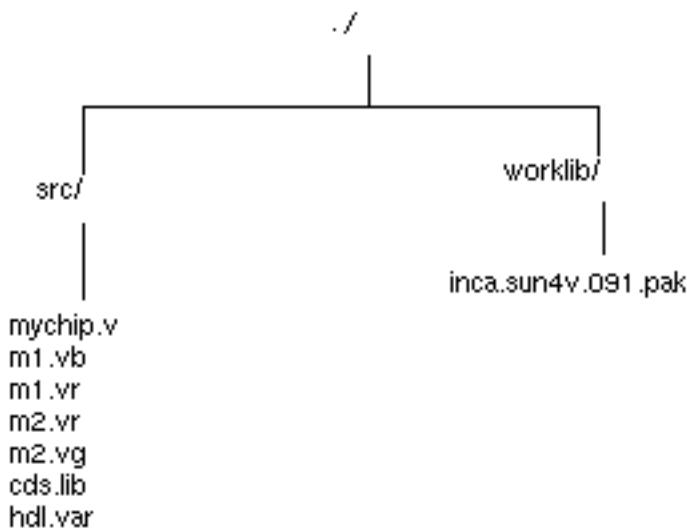
The file `mychip.v` gets compiled into the default module view. The design unit in Lib.Cell:View notation is `worklib.mychip:module`.

Each of the RTL representations, `m1.vr` and `m2.vr`, is compiled into its respective rtl view: `worklib.m1:rtl` and `worklib.m2:rtl`.

The behavioral representation of `m1`, described in the file `m1.vb`, is compiled into `worklib.m1:beh`.

The gate-level representation of `m2`, described in the file `m2.vg`, is compiled into `worklib.m2:gates`.

The library directory (`worklib`) contains one `.pak` file that contains all of the intermediate objects created by the compiler.



In this example, the top-level module is called `mychip`. To elaborate the design, specify this design unit on the *ncelab* command line using the Lib.Cell:View notation, as follows:

```
% ncelab worklib.mychip:module
```

Because there is only one view of module `mychip` in the library, the library and the view specification could be omitted, as follows:

```
% ncelab mychip
```

The elaborator generates a simulation snapshot for the design. Intermediate objects created during the elaboration phase are stored in the .pak file. The snapshot is also a Lib.Cell:View. In this example, the snapshot is called worklib.mychip:module. See [Chapter 8, “Elaborating the Design with ncelab,”](#) for details on how the elaborator names snapshots.

You can now invoke the *ncsim* simulator by specifying the name of the snapshot on the command line, as follows:

```
% ncsim worklib.mychip:module
```

Because there is only one snapshot in the library, the library and the view specification could be omitted, as follows:

```
% ncsim mychip
```

---

# Compiling Verilog Source Files with ncvlog

---

This chapter contains the following sections:

- [Overview](#)
- [ncvlog Command Syntax](#)
- [ncvlog Command Options](#)
- [Example ncvlog Command Lines](#)
- [hdl.var Variables](#)
  
- [Conditionally Compiling Source Code](#)
- [Controlling the Compilation of Design Units into Library.Cell:View](#)
- [Defining Macros on the Command Line](#)

## Overview

After writing or editing your Verilog source files, you have to compile them. The program that you use to analyze and compile your Verilog source is called *ncvlog*.

*ncvlog* performs syntactic and static semantic checking on the Verilog HDL design unit(s) (modules, macromodules, or UDPs). If no errors are found, compilation produces an internal representation for each HDL design unit in the source files. By default, these intermediate objects are stored in a single packed library database file in the work library directory.

In some cases, particularly if you are using a 5.x configuration file to control the binding of instances during elaboration, you must use the `-use5x` option (or the `NCUSE5X` variable in the `hdl.var` file) when you compile the Verilog source files. This option generates the packed library database file, but also creates the full Cadence 5.x library system, in which the intermediate objects for each design unit are stored in their own subdirectories under the work library, and three other files: `master.tag`, `verilog.v`, and `pc.db`. The full 5.x library structure and the additional files make it possible for tools that do not understand the default packed library structure to access the intermediate objects that are required by the tool.

If you are running the NC-Verilog simulator using the single-step invocation method (*ncverilog*), and only want to compile your source files, use the `+compile` option. This option stops the simulator after compilation.

If you are running the NC-Verilog simulator using multi-step invocation, invoke *ncvlog* with options and a Verilog source file name(s). These arguments can occur in any order except that parameters to options must immediately follow the option that they modify. You can also invoke *ncvlog* with the `-unit` or `-specificunit` option and a design unit name.

Examples:

```
% ncvlog [options] foo.v          ;# foo.v is a source file  
% ncvlog -unit worklib.mymod    ;# mymod is an HDL design unit
```

*ncvlog* treats each command-line argument that is not an option or a parameter to an option as a filename. For each filename, *ncvlog* first tries to open the file as specified. If this fails, each file extension that is specified with the `VERILOG_SUFFIX` variable is appended to the name, and *ncvlog* tries to open the file. The default file extension is `.v`. If no match is found, *ncvlog* tries the list of possible suffixes in the `hdl.var` variable `VIEW_MAP`. If all suffixes are exhausted, *ncvlog* generates an error.

For details on the `VERILOG_SUFFIX` and `VIEW_MAP` variables, see “[hdl.var Variables](#)” on [page 160](#).

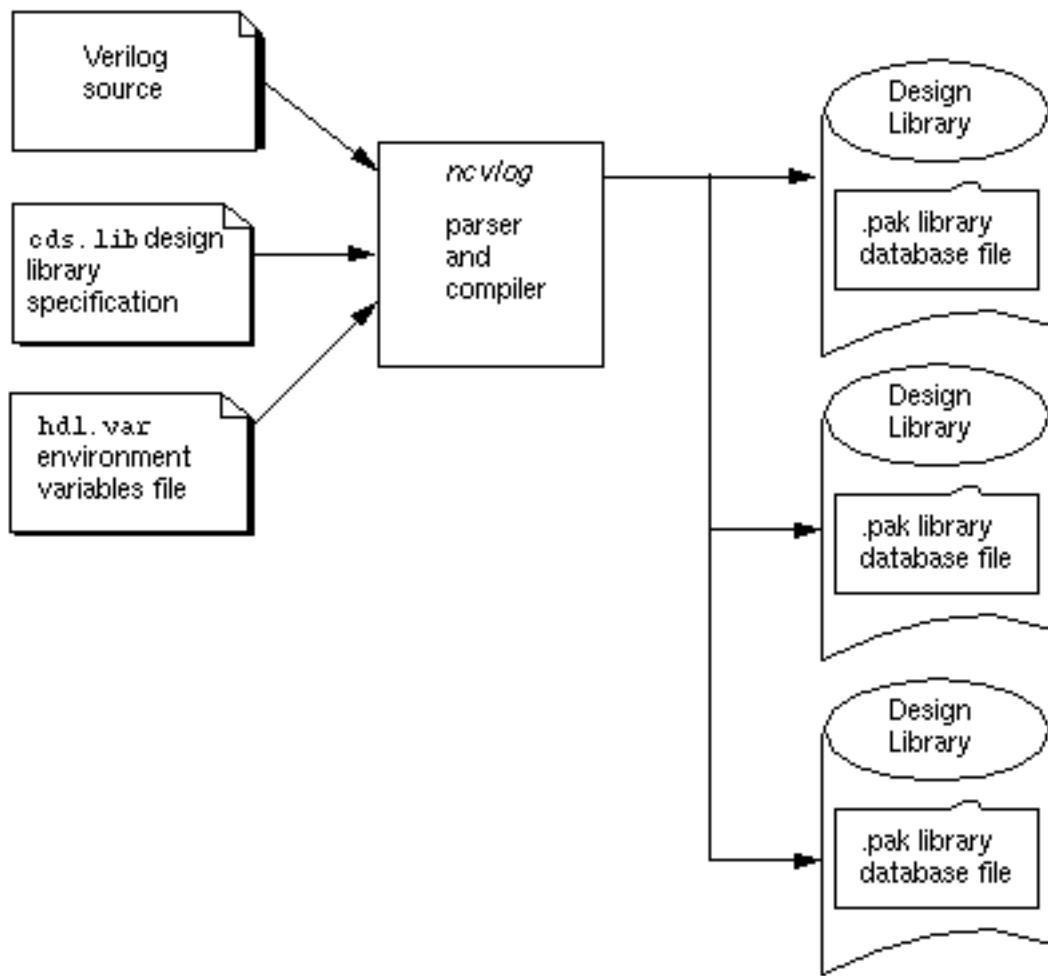
Design units are compiled into a Library.Cell:View. See “[The Library.Cell:View Approach](#)” on [page 109](#) for information on the NC-Verilog simulator library system.

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with *ncvlog*

The cell name is always set to the name of the design unit. You can control where the compiler stores compiled objects and the view names that are assigned to them. See “[Controlling the Compilation of Design Units into Library.Cell:View](#)” on page 165 for more information.

The following figure illustrates the *ncvlog* process flow:



When you change any of the design units in the hierarchy, you must recompile the design units that you have changed and re-elaborate the design hierarchy. You can automatically recompile all out-of-date design units in the hierarchy and re-elaborate the design by:

- Running *ncupdate*. This utility runs *ncvlog* to recompile any changed modules (and *ncvhdl* to recompile any changed VHDL units) and then runs *ncelab* to re-elaborate the design. *ncelab* also automatically invokes the *ncsdhc* utility to recompile the SDF source file if it detects a change in the file. The elaborator then generates a new snapshot. Use *ncupdate* when you want to update the snapshot, but do not want to simulate. See “[“ncupdate”](#) on page 928 for information on *ncupdate*.

- Including the `-update` option on the `ncsim` command. This option calls `ncupdate`, which recompiles any changed design units, recompiles the SDF file if necessary, re-elaborates the design, generates a new snapshot, and then invokes the simulator. Use `ncsim -update` if you want to update and simulate. See “[Updating Design Changes When You Invoke the Simulator](#)” on page 332 for more information.

## ncvlog Command Syntax

Command-line options can be abbreviated to the shortest unique string, indicated here with capital letters.

```
ncvlog [options] filename [filename ...]  
  
ncvlog [options] { -specificunit [lib.]cell[:view] filename |  
                  -unit [lib.]cell[:view] }  
  
[-Append log]  
[-CDslib cdslib_pathname]  
[-Checktasks]  
[-Define identifier[=value]]  
[-Errormax integer]  
[-File arguments_filename]  
[-HDLvar hdlvar_pathname]  
[-Help]  
[-IEee1364]  
[-INcdir directory]  
[-LExpragma]  
[-LIBcell]  
[-LINedebug]  
[-Logfile filename]  
[-Messages]  
[-NCError warning_code]  
[-NCFatal {warning_code | error_code}]  
[-NEverwarn]  
[-NOCopyright]  
[-NOLine]  
[-NOLog]  
[-NOMempack]  
[-NOPragmawarn]  
[-NOSTdout]  
[-NOWarn warning_code]
```

```
[-Pragma]
[-Specificunit [lib.]cell[:view] filename]
[-Status]
[-Unit [lib.]cell[:view]]
[-UPCase]
[-UPDate]
[-USe5x]
[-VErsion]
[-VIew view_name]
[-Work library]
```

## ncvlog Command Options

This section describes the options that you can use with the *ncvlog* command. Options can be entered in upper or lower case. Capital letters indicate the shortest possible abbreviation for an option.

### **-Append\_log**

Append log information from multiple runs of *ncvlog* to one log file. Use this option if you are going to run *ncvlog* multiple times to compile source files and you want all log information appended to one log file. If you do not use this option, the log file is overwritten each time you run *ncvlog*. For example, all log information for the following two runs of *ncvlog* is sent to the default log file (*ncvlog.log*):

```
% ncvlog -messages flop.v
% ncvlog -messages -append_log reg.v
```

If you use both **-append\_log** and **-nolog** on the command line, **-nolog** overrides **-append\_log**.

Because the log file is opened before variables in the *hdl.var* file are read, the **-append\_log** option is ignored with a warning if you define it with the **NCVLOGOPTS** variable in an *hdl.var* file.

### **-CDslib** *cdslib\_pathname*

Use the specified *cds.lib* file. See “[The cds.lib File](#)” on page 110 for details on the *cds.lib* file.

All tools and utilities that require a *cds.lib* file use a default search mechanism to find the *cds.lib* file. See “[The setup.loc File](#)” on page 131 for information on this search mechanism.

## **NC-Verilog Simulator Help**

### Compiling Verilog Source Files with ncvlog

---

Use the `-cdslib` option to override the default search order and force the compiler to use the specified `cds.lib` file.

Example:

```
% ncvlog -cdslib ~/design_lib/cds.lib test.v
```

The compiler reads the `cds.lib` file before it processes any variables defined in the `hdl.var` file. You cannot, therefore, include the `-cdslib` option with the `NCVLOGOPTS` variable in an `hdl.var` file.

### **-Checktasks**

Check for standard system tasks.

Use the `-checktasks` option to check for the presence of any non-predefined system tasks or functions in your source code. This includes checking for the presence of user-defined PLI tasks and functions. A warning message is generated for each task that is not a predefined task.

Example:

```
% ncvlog -checktasks top_mod.v
```

### **-Define *identifier*[=*value*]**

Define a variable name as:

- An empty text macro to be used for conditional compilation. For example,

```
-define structural
```

See “[Conditionally Compiling Source Code](#)” on page 163.

- A string. For example,

```
-define foo=2
```

See “[Defining Macros on the Command Line](#)” on page 178.

### **-Errormax *integer***

Abort after reaching the specified number of errors. By default, there is no limit on the number of error messages.

By using `-errormax`, you can limit the number of errors that are generated, fix those errors, and then rerun to check for other errors. This option is useful when you are compiling a large design that might contain numerous errors.

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with ncvlog

---

Example:

```
% ncvlog -errormax 10 source.v
```

#### **-File arguments \_filename**

Use the command-line arguments contained in the specified arguments file.

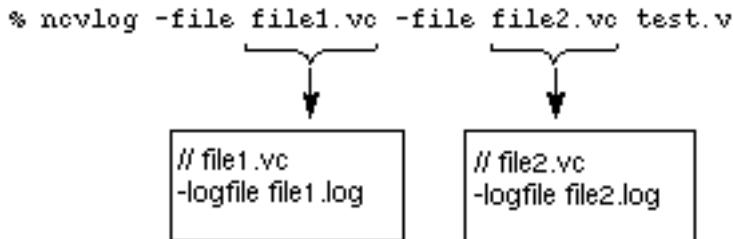
You can store frequently used or lengthy command lines by putting command-line arguments (command options and source file names) in a text file. When you invoke the compiler with the **-file** option, the arguments in the arguments file are incorporated with your command as if they had been entered on the command line.

The arguments file can contain command options, including other **-file** options, and source file names. The individual arguments within the arguments file must be separated by white space or comments. Both types of Verilog HDL comments are allowed.

Example:

```
% ncvlog -file ncvlog.args source.v
```

If you use the same command-line option with different arguments in different argument files, a fatal message is generated and compilation fails. For example, you cannot do the following:



**Note:** The arguments to a command option in an arguments file are treated as literal strings. Do not use quotation marks in the arguments unless you explicitly want them as part of the input. For example, use:

```
-define foo=16'h03
```

instead of

```
-define foo="16'h03"
```

which is the same as typing:

```
% ncvlog -define foo=\\"16'h03\\\"
```

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with ncvlog

---

You can also use the NCVLOGOPTS variable in an hdl.var file to include command-line options and source file names. For example, including -file args.vc, where the file args.vc contains the following text:

```
-messages  
adder.v  
adder_test.v
```

is the same as including the following in the hdl.var file:

```
DEFINE NCVLOGOPTS -messages adder.v adder_test.v
```

**Note:** The following options cannot be used in an hdl.var file: -cdslib, -hdlvar, -logfile. The following options are ignored if they are included in an hdl.var file: -help, -version, -nocopyright.

Example:

```
% ncvlog -file user.vc design.v
```

This command line uses the -file option to include a file that contains command-line arguments. The file user.vc is specific to the user and contains the following text:

```
/* This file contains my usual options */  
-file project1.vc      # Use the project-wide conventions contained in project1.vc  
-nocopyright          # Suppress printing of the copyright banner  
-messages              # Display informational and warning messages as well as error  
# messages  
-logfile design.log    # Send compiler output to design.log  
-errormax 10           # Abort after 10 error messages  
-file cpu.vc           # Include the arguments in cpu.vc
```

The file project1.vc is a central file of conventions for the project. It contains the following text:

```
/* Conventions for simulating project1 hardware */  
/* Larry Bird Widget Corp., Feb. 1996 */  
-ieee1364              # Check for compatibility with the IEEE 1364 standard  
-incdir ..../rtl_models # NC-Verilog will search this directory for include files  
-noline                 # Increase performance by not locating source lines on errors
```

The file cpu.vc in user.vc contains the following text:

```
cpu_netlist.v -file lib.vc      # Include the arguments in lib.vc and compile  
# the cpu design
```

The file lib.vc contains the following text:

```
array_lib_version2.v          # Gate array cell library  
joe_alu.v                     # Use Joe's alu description
```

### **-HDLvar *hdlvar\_pathname***

Use the specified `hdl.var` file. See “[The hdl.var File](#)” on page 118 for details on the `hdl.var` file.

All tools and utilities that require an `hdl.var` file use a default search mechanism to find the `hdl.var` file. See “[The setup.loc File](#)” on page 131 for information on this search mechanism. Use the `-hdlvar` option to override the default search order and force the compiler to use the specified `hdl.var` file.

Example:

```
% ncvlog -hdlvar ~/hdl.var test.v
```

You cannot include the `-hdlvar` option with the `NCVLOGOPTS` variable in an `hdl.var` file.

### **-HElp**

Display a list of the `ncvlog` command options with a brief description of each option.

```
% ncvlog -help
```

This option is ignored if you include it with the `NCVLOGOPTS` variable in an `hdl.var` file.

### **-IEee1364**

Check the source code for compatibility with the IEEE standard described in *IEEE-1364 Verilog Hardware Description Language Reference Manual*. For example, if your Verilog code contains attributes that are not part of the IEEE-1364 standard and you compile with the `-ieee1364` option to check for compatibility with the standard, a compliance warning message is generated.

Messages generated by the compiler contain references to relevant sections of the IEEE-1364 LRM.

Using this option is important if you are going to use other tools, such as a second simulator or a synthesis tool, that are compatible only with a particular standard or specification.

There are some compatibility checks that are performed during elaboration. The `-ieee1364` option can also be used when you invoke `ncelab`.

Example:

```
% ncvlog -ieee1364 source.v
```

### **-INcdir *directory***

Search the specified directory for include files.

Use the `include compiler directive to insert the contents of a source file into another source file. See the Verilog Language Reference Manuals for details on this compiler directive.

Use the -includir command-line option to specify the directories to search for files specified with the `include compiler directive.

Example:

```
% ncvlog -includir /design/tests top_mod.v
```

There is no limit to the number of -includir options you can specify.

The `include compiler directive works in the following way:

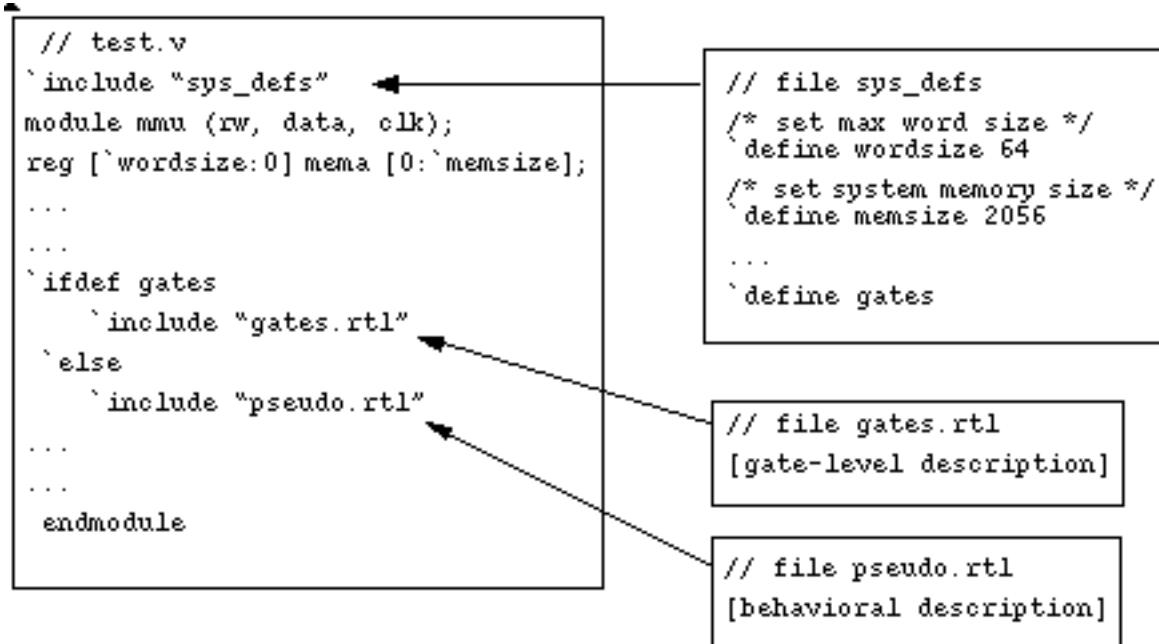
1. The compiler searches for the include file relative to the current working directory.
2. If the file is not found, the compiler searches the directories you specified with the -includir option. Directories are searched in the order you listed them on the command line, and the compiler uses the first file that it finds in the directory list.
3. If the compiler does not find the file in any of the directories you specified with the -includir option, an error is generated and compilation stops.

**Note:** Syntax errors in a `include file generate messages that usually display the correct file name and line number. However, wrong file names and line numbers may be displayed when some constructs are split across different file boundaries. These constructs include object or gate declarations, initial blocks, fork/join blocks, specify blocks, tasks, and functions.

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with ncvlog

The following example shows a source file that contains `include compiler directives.



In the example, the file `test.v` includes three `include compiler directives:

- The file `sys_defs` contains system definitions and is located in `../test_config`
- The file `gates.rtl` contains the gate-level description and is located in `../gates`
- The file `pseudo.rtl` contains the behavioral description and is located in `../pseudo`

The ``ifdef` statement in `test.v` is included for conditional compilation. It specifies that if the macro `gates` is defined, the file `gates.rtl` is to be included; otherwise, the file `pseudo.rtl` is to be included. Macro `gates` is defined in the file `sys_defs` with the ``define` compiler directive. For this simulation, the compiler must be able to find files `sys_defs` and `gates.rtl`. Use the `-incdir` option to specify the directories to search for these files.

You would compile the model with the following command:

```
% ncvlog -incdir ../test_config -incdir ../gates test.v
```

## **-LExpragma**

Enable processing of lexical pragmas.

Lexical pragmas are pragmas that can be associated with any Verilog or VHDL construct to indicate that translation/synthesis is turned off. The following pragmas are classified as lexical pragmas:

- `cadence translate_off` and `cadence translate_on` (also: `synopsys translate_off` and `synopsys translate_on`)
- `cadence synthesis_off` and `cadence synthesis_on` (also: `synopsys synthesis_off` and `synopsys synthesis_on`)
- `rtl_synthesis off` and `rtl_synthesis on`

If you compile with the `-lexpragma` option, any HDL constructs between a `translate_off/synthesis_off` pragma and a `translate_on/synthesis_on` pragma are treated as comments. For example, if the source code contains the following pragmas, `'define CI2CLKP 10` is treated as a comment.

```
'define CI2CLKP 512
// cadence translate_off
'define CI2CLKP 10
// cadence translate_on
```

If you use both `-pragma` and `-lexpragma`, lexical pragmas are processed with `-lexpragma`.

## **-LIBcell**

Insert `'celldefine` and `'endcelldefine` compiler directives to tag module instances as cell instances. Cells are used by certain PLI and VPI routines for applications such as delay calculation.

Use this option if you want to precompile a library and mark all cells with `'celldefine`.

**Note:** Verilog-XL automatically tags cells extracted from libraries as cells, and you can use the `+nolibcell` option to override this. The NC-Verilog simulator does not automatically do this, and the `-libcell` option must be used to insert the compiler directives. However, if you are running the NC-Verilog simulator using the `ncverilog` command, the behavior is identical to Verilog-XL. See [Chapter 4, “Running NC-Verilog with the ncverilog Command,”](#) for information on `ncverilog`.

### **-LINedebug**

Enable support for setting line breakpoints and for single-stepping through code.

By default, models are compiled with performance optimizations turned on. Some of these optimizations disable the ability to set line breakpoints and to single-step though code. Use `-linedebug` to override this behavior.

Using this option sets the default access to simulation objects to read/write/connectivity when the design is elaborated. Do not use this option if you want to run in regression mode. See [“Enabling Read, Write, or Connectivity Access to Simulation Objects”](#) on page 248 for more information.

### **-LOGfile *filename***

Use the specified name for the log file instead of the default name `ncvlog.log`.

Example:

```
% ncvlog -logfile counter.log counter.v
```

Use `-nolog` if you don't want a log file. If you use both `-logfile` and `-nolog` on the command line, `-logfile` overrides `-nolog`.

Use `-append_log` if you are going to run `ncvlog` multiple times to compile source files and you want all log information appended to one log file. If you do not use this option, the log file is overwritten each time you run `ncvlog`.

Because the log file is opened before variables in the `hdl.var` file are read, the `logfile` option is ignored with a warning if you define it with the `NCVLOGOPTS` variable in an `hdl.var` file.

### **-Messages**

Print informative messages during execution.

Example:

```
% ncvlog -messages source.v
```

By default, compiler messages are printed to a log file called `ncvlog.log`. Use `-logfile` to rename the log file. Use `-nolog` if you don't want a log file.

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with ncvlog

---

Messages are also printed to the screen by default. Use `-nostdout` if you want to suppress printing to the screen.

#### **`-NCError warning_code`**

Increase the severity level of the specified warning message from warning to error. The `warning_code` argument is the message code (mnemonic) that appears in the warning message following the severity code.

Example:

```
% ncvlog -ncerror CUVWSP source.v
```

You can include multiple `-ncerror` options on the command line.

Using this option can change the behavior of the tool because functions that return errors instead of warnings may behave differently. Warnings that are changed to errors are counted in the error count limit that you specify with the `errormax` option.

#### **`-NCFatal {warning_code | error_code}`**

Increase the severity level of the specified warning message or error message from warning or error to fatal. The `warning_code` or `error_code` argument is the message code (mnemonic) that appears in the message following the severity code.

Example:

```
% ncvlog -ncfatal DLCPTH source.v
```

You can include multiple `-ncfatal` options on the command line.

#### **`-NEverwarn`**

Disable printing of all warning messages.

Example:

```
% ncvlog -neverwarn source.v
```

To turn off one or more specific warning messages, use `nowarn`.

### **-NOCopyright**

SUPPRESS printing of the copyright banner.

Because the copyright banner is displayed before any variables in the `hdl.var` file are processed, this option is ignored if you include it with the `NCVLOGOPTS` variable in an `hdl.var` file.

### **-NOLine**

DO NOT locate source line on errors.

Use the `-noline` option to stop the compiler from scanning source files when it finds errors.

The compiler uses a small buffer to store and keep track of the most recently read source lines. Occasionally, `ncvlog` refers back to this buffer to print error messages. If an error occurs and the source line is not contained in the buffer, `ncvlog` reopens the source file and scans for the line it needs. Using the `-noline` option can improve performance by eliminating this source file scanning.

Example:

```
% ncvlog -noline source.v
```

### **-NOLOG**

DO NOT generate a log file. By default, `ncvlog` generates a log file called `ncvlog.log`.

If you use both `-nolog` and `-logfile` on the command line, `-logfile` overrides `-nolog`.

### **-NOMempack**

Design units must be compiled with this option to access memory array values using the PLI routine `tf_nodeinfo()`. If you do not compile with this option and call `tf_nodeinfo()` to access memory array values, the following error message is generated, and the `expr_value_p` field of the `s_tfexprinfo` structure will be NULL.

`tf_nodeinfo()` called on a memory which was compiled without the `ncvlog -NOMEMPACK` option. No access to value.

In situations where `tf_nodeinfo()` is called but the value is not accessed, the application continues running despite the error message.

### **-NOPragmawarn**

Do not display warning messages related to pragmas.

### **-NOStdout**

Suppress printing of output to the screen.

The `-nostdout` option does not change what is written to the log file.

### **-NOWarn *warning\_code***

Disable printing of the warning with the specified code. The *warning\_code* argument is the message code (mnemonic) that appears in the warning message following the error severity code.

Example:

```
% ncvlog -nowarn INTIVF source.v
```

You can include multiple `-nowarn` options on the command line.

### **-Pragma**

Parse pragmas contained in the HDL source files.

Some pragmas related to translation or synthesis control, such as `cadence synthesis_off` and `cadence translation_off`, are classified as *lexical* pragmas. These pragmas can be associated with any Verilog or VHDL construct to indicate that translation/synthesis is turned off. There are restrictions on the use of these pragmas if you compile with the `-pragma` option. Specifically, these pragmas do not get applied if they are used:

- Around `'define`, `'include`, or `'ifdef` constructs
- On the branches of `if` or `case` constructs

You can include the `-lexpragma` option to enable the processing of these lexical pragmas.

If you use both `-pragma` and `-lexpragma`, lexical pragmas are processed with `-lexpragma`.

### **-S**Pecificunit [*lib.]cell[:view]* *filename*

Compile only the specified design unit in the source file.

Use the `-specificunit` option to compile one design unit from a source file that contains multiple design units.

Examples:

```
% ncvlog -specificunit arith alu.v  
% ncvlog -specificunit worklib.arith alu.v
```

If you specify the library, the design unit is compiled into that library, unless a library is specified with the ``worklib` compiler directive. If you specify the view, that view name is used, unless a view is specified with the ``view` compiler directive. See [“Controlling the Compilation of Design Units into Library.Cell:View”](#) on page 165 for more information.

Macros and compiler directives that are defined in the specific unit that you are compiling are not inherited by other units in the source file.

### **-S**tatus

Print statistics on memory and CPU usage after compilation.

The following example shows the output of the `-status` option:

```
ncvlog: Memory Usage - 2.8M program + 0.8M data = 3.6M total  
ncvlog: CPU Usage - 0.5s system + 0.2s user = 0.7s total (0.6s, 100.0% cpu)
```

### **-U**nit [*lib.]cell[:view]*

Recompile the source file that contains the specified design unit.

Use the `-unit` option to recompile a specific design unit that has been compiled previously and that you have subsequently edited.

The argument to the `-unit` option is the name of the design unit. You cannot specify a source file name. The compiler automatically finds and uses the correct files.

Examples:

```
% ncvlog -unit arith  
% ncvlog -unit worklib.arith
```

### **-UPCase**

Convert all lowercase letters in identifiers to uppercase, except for text within strings, so that a Verilog source description becomes case insensitive.

This option provides backward compatibility with the Verilog-XL `-u` option, and is intended to solve problems that can potentially arise when part of a design description may refer to an object using lowercase letters (for example, `sum`) while another part of the description, typically the output of a converter or netlister, may refer to the same object in uppercase or mixed case letters (`SUM`, for example).

Identifiers are case sensitive in Verilog. For example, if you have a signal called `Sum` and you compile with `-upcase`, you must use `SUM` when you refer to the signal.

Example:

```
% ncvlog -upcase source.v
```

**Note:** Using this option may cause problems with name mapping in a mixed-language simulation.

### **-UPDate**

Do not write a new intermediate representation for a module if the module is up-to-date.

Use this option to recompile the design after adding a design unit, a source file, or compiler directives to the design, or if you change a design unit in a way that introduces a new cross-file dependency. In these cases, the `ncupdate` utility or `ncsim -update` will not update correctly.

If you are running the NC-Verilog simulator using the `ncverilog` command, this option is on by default. Use the `+noupdate` option to override it. See [Chapter 4, “Running NC-Verilog with the ncverilog Command,”](#) for information on `ncverilog`.

### **-USe5x**

Create the full Cadence 5.x library structure when compiling the Verilog source files.

By default, the compiler stores all intermediate objects for all design units in one packed library database file in the work library directory. The `-use5x` option (or the `NCUSE5X` variable in the `hdl.var` file) generates the packed library database file, but also creates the full Cadence 5.x library structure, in which the intermediate objects for each design unit are

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with ncvlog

---

stored in their own subdirectories under the work library. It also creates three other files for each design unit: `master.tag`, `Verilog.v`, and `pc.db`.

The full 5.x library structure and the additional files make it possible for tools that do not understand the default packed library structure to access the intermediate objects that are required by the tool. For example, you must compile the source files with the `-use5x` option if you want to use a 5.x configuration file to control the binding of instances during elaboration.

#### **-VErsion**

Print the version of `ncvlog` and exit.

This option is ignored if you include it with the `NCVLOGOPTS` variable in an `hdl.var` file.

#### **-View *view\_name***

Use the specified name as the view name for the compiled design unit. This option is used to create different view names for different representations of a design unit with the same name.

Using this option overrides any setting for the `VIEW_MAP` or `VIEW` variables in the `hdl.var` file.

Examples:

```
% ncvlog -view rtl source.v  
% ncvlog -view gates source.v
```

The `-view` option is used only for specifying a view name for the compiled design unit. To recompile a specific view, use `-unit lib.cell:view`.

See “[Controlling the Compilation of Design Units into Library.Cell:View](#)” on page 165 for more information on specifying a view name.

#### **-Work *library***

Use the specified library as the *work* library.

The work library is the library that the compiler uses to store compiled objects (Verilog modules, macromodules, and UDPs) and other derived data. Your work library is usually defined in the `hdl.var` file with the `LIB_MAP` or `WORK` variable. Use the `-work` option to override the current `hdl.var` setting.

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with ncvlog

---

Example:

```
% ncvlog -work designlib source.v
```

**Note:** The library logical name must be defined in the `cds.lib` file. See “[The cds.lib File](#)” on page 110 for details on the `cds.lib` file.

See “[Controlling the Compilation of Design Units into Library.Cell:View](#)” on page 165 for more details on how to control where the compiler stores compiled design objects.

## Example ncvlog Command Lines

The following command includes the `-messages` option, which prints compiler messages.

```
% ncvlog -messages 2bit_adder_test.v
ncvlog: v1.0.(p1): (c) Copyright 1995, Cadence Design Systems, Inc.
file: 2bit_adder_test.v
    module worklib.top
        errors: 0, warnings: 0
%
```

The following example uses the `-work` option to define the current working library as `aludesign`. This overrides the definition of the `WORK` variable in the `hdl.var` file.

```
% ncvlog -work aludesign 2bit_adder_test.v
```

The following example uses the `-file` option to include a file called `ncvlog.vc`, which includes a set of command-line options, such as `-messages`, `-nocopyright`, `-errormax`, and `-inmdir`.

```
% ncvlog -file ncvlog.vc 2bit_adder_test.v
```

In the following example, the `-ieee1364` option checks for compatibility with the IEEE specification. Error messages reference the IEEE Language Reference Manual.

```
% ncvlog -ieee1364 2bit_adder_test.v
```

The following example includes the `-inmdir` option to specify a directory to search for include files.

```
% ncvlog -inmdir ~larrybird/bigdesign 2bit_adder_test.v
```

In the following example, `-errormax 10` tells the compiler to abort after 10 errors. The `-noline` option suppresses the reporting of source lines when errors are encountered. Using this option can improve performance when compiling very large source files that contain errors.

```
% ncvlog -errormax 10 -noline 2bit_adder.v
```

The following example includes the `-logfile` option to send output to a log file called `adder.log` instead of to the default log file `ncvlog.log`.

```
% ncvlog -messages -logfile adder.log 2bit_adder.v
```

The following example illustrates how to use `-nowarn` to suppress the printing of a specific warning message.

```
% ncvlog *.v
ncvlog: v3.00.(p1): (c) Copyright 1995 - 1999 Cadence Design Systems, Inc.
ncvlog: *W,HVLOGF: the -LOGFILE option can not be specified in an hdl.var file.
file: board.v
```

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with ncvlog

---

```
module worklib.board
    errors: 0, warnings: 0
file: clock.v
    module worklib.m555
        errors: 0, warnings: 0
...
...
...
Total errors/warnings found outside modules and primitives:
    errors: 0, warnings: 1
% ncvlog -nowarn hvlogf *.v
ncvlog: v3.00.(p1): (c) Copyright 1995 - 1999 Cadence Design Systems, Inc.
file: board.v
    module worklib.board
        errors: 0, warnings: 0
file: clock.v
    module worklib.m555
        errors: 0, warnings: 0
...
...
...
Total errors/warnings found outside modules and primitives:
    errors: 0, warnings: 1
```

The following example includes the `-linedebug` option, which disables the optimizations that prevent line debug capabilities.

```
% ncvlog -linedebug 2bit_adder.v
```

## hdl.var Variables

The following `hdl.var` variables are used by *ncvlog*.

**Note:** See “[The hdl.var File](#)” on page 118 for information about the `hdl.var` file.

### ■ LIB\_MAP

This variable maps files and directories to the names of libraries where you want them to be compiled. Use the plus sign ( + ) to create a default for files or directories that are not explicitly specified.

Example:

```
DEFINE LIB_MAP ( \
    ./design/lib1/... => lib1, \
    ./source          => lib2, \
    top.v            => lib3, \
    +                => worklib )
```

In this example:

- ❑ All files and directories below `./design/lib1` are mapped to library `lib1`.
- ❑ All files in the `./source` directory are mapped to library `lib2`.
- ❑ The file `top.v` is mapped to library `lib3`.
- ❑ Everything else is mapped to library `worklib`.

### ■ NCUSE5X

This variable generates the packed library database file, but also creates the full Cadence 5.x library system, in which the intermediate objects for each design unit are stored in their own subdirectories under the work library. It also creates three other files for each design unit: `master.tag`, `verilog.v`, and `pc.db`.

The full 5.x library structure and the additional files make it possible for tools that do not understand the default packed library structure to access the intermediate objects that are required by the tool. For example, you must compile the source files with this variable defined (or by using the `-use5x` option on the `ncvlog` command line) if you want to use a 5.x configuration file to control the binding of instances during elaboration.

### ■ NCVLOGOPTS

The value of this variable is taken as additional arguments to the `ncvlog` command. When there is a conflict, the value on the command line overrides the value in `NCVLOGOPTS`.

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with ncvlog

---

Example:

```
DEFINE NCVLOGOPTS -messages -errormax 10
```

#### ■ SRC\_ROOT

This variable lets you define an ordered list of paths to search for source files when you are updating a design, either by running `ncsim -update` or by rerunning `ncverilog`. The paths listed in the definition of this variable tell the NC tools where to look for source files if design units are out-of-date.

**Note:** The paths listed in the `SRC_ROOT` variable do not tell the parser where to find the source files that you specify on the command line. This variable is used to help the NC tools to find source files after initial compilation. You must define the `SRC_ROOT` variable before the initial compilation.

Example:

Suppose that your project has a centralized release area for source files named `/project/source`, and that you have a local “check out” area called `~johndoe/source`. In your `hdl.var` file, you can define the `SRC_ROOT` variable as follows:

```
DEFINE SRC_ROOT (~johndoe/source, /project/source)
```

This definition of `SRC_ROOT` tells the tools to look first in `~johndoe/source` for the source files that contain the out-of-date units when you update the design.

For example, suppose that the `/project/source` area contains a source file called `sourcefile.v`. While simulating in your own local area, `~johndoe/source`, you notice a bug in `/project/source/sourcefile.v`. Using your CM system, you check out `sourcefile.v`. This copies the file into `~johndoe/source`. You then edit `sourcefile.v`. You now have to update the snapshot because of this change.

When you update the design with `ncsim -update` or rerun `ncverilog`, the program detects that some design units are out-of-date. The definition of `SRC_ROOT` tells the tools to look first in `~johndoe/source` for the source files that contain the out-of-date units. In this example, the simulator uses `sourcefile.v` in `~johndoe/source` to recompile the out-of-date design units. When you are satisfied with the simulation results, you can check the source file back into the central release area.

Given the same definition of the `SRC_ROOT` variable shown above, any subsequent update that you do that includes `sourcefile.v` (for example, another designer may have edited the file) will use the `sourcefile.v` in your local area. Delete or rename the file in your local area after checking it back in to the central release area.

#### ■ VERILOG\_SUFFIX

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with ncvlog

---

This variable specifies valid file extensions for Verilog source files. *ncvlog* treats each command-line argument that is not an option or a parameter to an option as a filename. It first tries to open the file as specified. If this fails, each file extension specified with the VERILOG\_SUFFIX variable is appended to the name, and *ncvlog* tries to open the file. If no match is found, *ncvlog* tries the list of possible suffixes in the hdl.var variable VIEW\_MAP. If all suffixes are exhausted, an error is generated.

You can specify a single value or a list of values. For example:

```
DEFINE VERILOG_SUFFIX .vlog  
DEFINE VERILOG_SUFFIX (.vlog, .vsyn, .vrtl, .vgate)
```

#### ■ **VIEW**

This variable determines the view name. If you invoke *ncvlog* without the -view option, the parser searches for this variable. If the -view option is set, *ncvlog* uses the value as the view name.

```
DEFINE VIEW rtl
```

#### ■ **WORK**

This variable defines the work library into which the HDL design units are compiled.

```
DEFINE WORK worklib
```

#### ■ **VIEW\_MAP**

This variable maps file extensions to view names.

In the following example, files with a .rtl extension are given a view name of rtl.

```
DEFINE VIEW_MAP ( .v      => behav, \  
                  .rtl   => rtl, \  
                  .gate  => gate \  
                  .vs    => module )
```

## Conditionally Compiling Source Code

Use the conditional compilation compiler directives (``ifdef`, ``else`, and ``endif`) to conditionally include lines of a Verilog HDL source description during compilation. The ``ifdef`` compiler directive checks whether a variable name is defined either in the source code or on the command line. If the variable name is defined, the compiler includes the lines in the source description.

There are two ways to define `ifdef variables:

- Use the `define compiler directive. For example:

```
'define debug
```

See the Verilog Language Reference Manuals for details on this compiler directive.

- Use the -define command-line option.

To control conditional compilation, define a variable name as an empty text macro. The syntax is:

```
-define text_macro_name
```

For example,

```
-define debug  
-define sun3
```

You can have multiple -define arguments on a command line. For example:

```
-define sun4 -define structural
```

If you define the same macro name differently using a `define compiler directive and a -define command-line option, the command-line option overrides the compiler directive.

The compiler does not check the syntax for any ignored group of lines. However, even though *ncvlog* does not check the syntax of this text, it must conform to the lexical conventions for white space, comments, numbers, strings, identifiers, keywords, and operators.

The following example shows you how to define a macro on the command line with the -define option.

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with ncvlog

---

```
% more test.v                      ;# Display the source file.
module test;
initial
begin
'ifdef debug      // If debug is defined, execute the following line.
    $display("debug is defined.");
'else           // If debug is not defined, execute the following line.
    $display("debug is not defined.");
'endif
end
endmodule

% ncvlog -mess test.v    ;# Compile with ncvlog. Macro debug is not defined in
                        ;# the model or on the command line.
ncvlog: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
file: test2.v
module worklib.test
errors: 0, warnings: 0
% ncelab test                  ;# Elaborate with ncelab.
ncelab: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
% ncsim test                   ;# Invoke the simulator.
ncsim: *W,BATONL: Only batch mode implemented, forcing -BATCH option.
ncelab: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
ncsim> run        ;# Run the simulation. Model displays "debug is not defined."
debug is not defined
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
% ncvlog -mess -define debug test.v    ;# Compile with ncvlog. Macro debug is
                        ;# defined on the command line.
ncvlog: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
file: test2.v
module worklib.test
errors: 0, warnings: 0
% ncelab test
ncelab: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
% ncsim test
ncsim: *W,BATONL: Only batch mode implemented, forcing -BATCH option
ncsim: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems
ncsim> run        ;# Run the simulation. Model displays "debug is defined."
debug is defined
ncsim: *W,RNQUIE: Simulation is complete.
```

## Controlling the Compilation of Design Units into Library.Cell:View

When you run the *ncvlog* compiler, your Verilog HDL design units (modules, macromodules, and UDPs) are compiled into a Library.Cell:View. The cell name is always set to the name of the design unit. You can control where the compiler stores compiled objects and the view names that are assigned to them.

To specify the library and view, you can use variables defined in the `hdl.var` file, command-line options, or compiler directives.

The order of precedence is as follows:

- [The Default](#)
- The definitions of the `LIB_MAP` and `VIEW_MAP` variables in the `hdl.var` file. See “[The LIB\\_MAP and VIEW\\_MAP Variables](#)” on page 167.
- The definitions of the `WORK` and `VIEW` variables in the `hdl.var` file. See “[The WORK and VIEW Variables](#)” on page 169.
- The `-work` or `-view` command-line options. See “[The -work and -view Options](#)” on page 171.
- The `-specificunit` command-line option with a library and/or view specification. See “[The -specificunit Option](#)” on page 172.
- The ``worklib` and ``view` compiler directives. See “[The `worklib and `view Compiler Directives](#)” on page 173.

See “[Mapping of Modules Defined Within `include Files](#)” on page 175 for information on the library and view mapping of modules defined within ``include` files.

### The Default

Design units are compiled into the specified work library. A work library must be defined or the compiler generates a fatal error message.

The work library can be defined using:

- Variables defined in the `hdl.var` file (`LIB_MAP` or `WORK`)
- The `-work` command-line option
- The ``worklib` compiler directive

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with ncvlog

---

If the view name is not explicitly specified, the default is to use the name of the kind of design unit being processed. This is:

- module for modules and macromodules
- udp for UDPs

In the following example, the work library is defined as `worklib` in the `hdl.var` file by defining the `WORK` variable. No view mapping is specified.

```
;# Use ncelp to display the hdl.var file. Line 5 in the hdl.var file defines
;# the work library as worklib. No view mapping is specified.
% ncelp -hdlvar
ncelp: v1.2.(b1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
Parsing -HDLVAR file /usr1/larrybird/hdl.var.
hdl.var files:
1: /usr1/larrybird/hdl.var
Variables defined:
Defined in /usr1/larrybird/hdl.var:
Line # Name      Value
-----  -----
1      NCVLOGOPTS -messages
2      NCELABOPTS -messages
3      NCSIMOPTS -messages
5      WORK       worklib

;# Compile the source files. No view mapping is specified on the command line
;# or in the source files. Modules are compiled into
;# worklib.module_name:module. For example, module board is compiled into
;# worklib.board:module.
% ncvlog *.v
ncvlog: v1.2.(b1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
file: board.v
    module worklib.board
        errors: 0, warnings: 0
file: clock.v
    module worklib.m555
        errors: 0, warnings: 0
file: counter.v
    module worklib.m16
        errors: 0, warnings: 0
file: ff.v
    module worklib.dEdgeFF
        errors: 0, warnings: 0
```

## The LIB\_MAP and VIEW\_MAP Variables

The first place that the compiler searches to find information on which library to compile source files into and what view names to use is the `hdl.var` file. It first looks for definitions of the `LIB_MAP` and `VIEW_MAP` variables.

`LIB_MAP` maps files and directories to library names. There are four mapping types that you can specify:

```
DEFINE LIB_MAP ( directory => library, \
                 directory/... => library, \
                 file => library, \
                 + => library )
```

Use the plus sign (+) to specify files or directories that are not explicitly stated.

For example, a `LIB_MAP` definition of:

```
DEFINE LIB_MAP ( ./design => designlib, \
                 ./source/lib1/... => lib1, \
                 myfile.v => mylib, \
                 + => worklib )
```

causes:

- Any file in the directory `./design` to be compiled into `designlib`.
- All files and directories below `./source/lib1` to be compiled into `lib1`.
- The file `myfile.v` to be compiled into `mylib`.
- Any other file to be compiled into `worklib`.

`VIEW_MAP` maps file extensions to view names. The syntax is:

```
DEFINE VIEW_MAP ( file_extension => view_name, ... )
```

Use the plus sign (+) to specify a default view mapping.

For example, a `VIEW_MAP` definition of:

```
DEFINE VIEW_MAP ( .v => behav, .r => rtl, .g => gate, + => module)
```

causes files with `.v` file extensions to create views with the name `behav`, `.r` files to create views with the name `rtl`, and `.g` files to create views with the name `gate`. Any other file extensions map to a view called `module`.

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with ncvlog

---

The LIB\_MAP and VIEW\_MAP variables are also used by the elaborator to resolve instances. See “[How Modules and UDPs Are Resolved During Elaboration](#)” on page 233 for more information.

In the following example, the LIB\_MAP and VIEW\_MAP variables are defined in the hdl.var file to determine which library the design units are compiled into and which view names they get.

```
;# Use ncelp to display the hdl.var file. Line 5 defines the LIB_MAP variable.
;# Files in the board/ directory will be compiled into designlib. Line 8 defines
;# the VIEW_MAP variable. Files with a .v extension will get a view name of
;# behav. Files with a .rtl extension will get a view name of rtl.

% ncelp -hdlvar
ncelp: v1.2.(b1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
Parsing -HDLVAR file /usr1/cousy/hdl.var.

hdl.var files:
1: /usr1/cousy/hdl.var

Variables defined:
Defined in /usr1/cousy/hdl.var:

Line # Name          Value
-----  -----
1      NCVLOGOPTS   -messages
2      NCELABOPTS   -messages
3      NCSIMOPTS    -messages
5      LIB_MAP       ( /usr1/cousy/inca/board => designlib , + => worklib )
8      VIEW_MAP      ( .v  => behav , .rtl => rtl , .gate => gate , + => module )

;# Source files are in the board/ directory. Compile all .v source files.
;# Modules are compiled into designlib.module_name:behav. For example, module
;# board is compiled into designlib.board:behav.

% ncvlog *.v
ncvlog: v1.2.(b1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.

file: board.v
    module designlib.board:behav
        errors: 0, warnings: 0
file: clock.v
    module designlib.m555:behav
        errors: 0, warnings: 0
file: ff.v
    module designlib.dEdgeFF:behav
        errors: 0, warnings: 0
```

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with ncvlog

---

```
# Compile all .rtl source files. Modules are compiled into
## designlib.module_name:rtl. For example, module m16 is compiled into
## designlib.m16:rtl.

% ncvlog *.rtl
ncvlog: v1.2.(b1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
file: counter.rtl
    module designlib.m16:rtl
        errors: 0, warnings: 0
```

## The WORK and VIEW Variables

If the compiler does not find definitions of the LIB\_MAP and VIEW\_MAP variables, it looks for definitions of the WORK and VIEW variables.

The WORK variable defines the work library. All design units are compiled into the specified library. For example, if you have the following definition in your hdl.var file, design units are compiled into the library called worklib.

```
DEFINE WORK worklib
```

The VIEW variable defines the view name. All design units receive the same view name. For example, if you have the following definition in your hdl.var file, design units are given a view name of behav.

```
DEFINE VIEW behav
```

If both LIB\_MAP/VIEW\_MAP variables and WORK/VIEW variables are defined, the definition of WORK overrides LIB\_MAP, and the definition of VIEW overrides VIEW\_MAP.

**Note:** If the WORK variable and the LIB\_MAP variable are both defined, the definition of the WORK variable overrides the definition of the LIB\_MAP variable. This is important to remember if you are doing a mixed-language simulation, where you might define the LIB\_MAP variable to control where the Verilog design units get compiled and then define the WORK variable to define the work library for VHDL. In this case, all design units would be compiled into the library specified with the WORK variable. To avoid this, you can either remove the definition of the WORK variable and then use the -work command-line option when compiling VHDL, or you can remove the definition of the LIB\_MAP variable and then use the -work option when compiling Verilog.

In the following example, the WORK and VIEW variables are defined in the hdl.var file.

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with ncvlog

---

```
# Use ncelp to display the hdl.var file. Line 5 defines the WORK variable.
# All files will be compiled into worklib. Line 6 defines the VIEW variable.
# All files will be compiled with a view name of gates.

% ncelp -hdlvar
ncelp: v1.2.(b1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
Parsing -HDLVAR file /usr1/jordan/hdl.var.
hdl.var files:
1: /usr1/jordan/hdl.var

Variables defined:
Defined in /usr1/jordan/hdl.var:
Line # Name      Value
-----  -----
1      NCVLOGOPTS -messages
2      NCELABOPTS -messages
3      NCSIMOPTS -messages
5      WORK       worklib
6      VIEW       gates

# Compile all .vg source files. Modules are compiled into worklib. All modules
# get a view name of gates.

% ncvlog *.vg
ncvlog: v1.2.(b1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
file: board.vg
    module worklib.board:gates
        errors: 0, warnings: 0
file: clock.vg
    module worklib.m555:gates
        errors: 0, warnings: 0
file: counter.vg
    module worklib.m16:gates
        errors: 0, warnings: 0
file: ff.vg
    module worklib.dEdgeFF:gates
        errors: 0, warnings: 0
```

## The **-work** and **-view** Options

If the compiler does not find definitions of the `LIB_MAP` and `VIEW_MAP` variables or definitions of the `WORK` and `VIEW` variables, it looks for the `-work` and `-view` command-line options.

The argument to `-work` is a library name.

The argument to `-view` is a view name.

Using the command-line options overrides any variable definitions in the `hdl.var` file.

In the following example, the `-work` and `-view` command-line options are used to override the variable definitions in the `hdl.var` file.

```
;# Use ncelp to display the hdl.var file. Line 5 defines the LIB_MAP variable.  
;# Files in the board directory will be compiled into designlib. Line 8 defines  
;# the VIEW_MAP variable. Files with .v extension will be compiled with a view  
;# name of behav.  
% ncelp -hdlvar  
ncelp: v1.2.(b1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.  
Parsing -HDLVAR file /usr1/mchale/hdl.var.  
hdl.var files:  
1: /usr1/mchale/hdl.var  
Variables defined:  
Defined in /usr1/mchale/hdl.var:  
Line # Name Value  
-----  
1 NCVLOGOPTS -messages  
2 NCELABOPTS -messages  
3 NCSIMOPTS -messages  
5 LIB_MAP ( /usr1/mchale/inca/board => designlib , + => worklib )  
8 VIEW_MAP ( .v => behav , .rtl => rtl , .gate => gate , + => module )  
  
;# Compile all .v source files. Use -work and -view to override the mapping  
;# specified in the hdl.var file. Modules are compiled into worklib with a view  
;# name of gate.  
% ncvlog -work worklib -view gate *.v  
ncvlog: v1.2.(b1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.  
file: board.v  
    module worklib.board:gate  
        errors: 0, warnings: 0  
file: clock.v  
    module worklib.m555:gate
```

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with ncvlog

---

```
    errors: 0, warnings: 0
file: counter.v
    module worklib.m16:gate
        errors: 0, warnings: 0
file: ff.v
    module worklib.dEdgeFF:gate
        errors: 0, warnings: 0
```

## The **-specificunit** Option

If the compiler does not find definitions of the LIB\_MAP and VIEW\_MAP variables, definitions of the WORK and VIEW variables, or the -work and -view command-line options, it looks for the -specificunit command-line option. This option is used to compile one design unit from a source file that contains multiple design units, and takes [library.]cell[:view] as an argument.

- If you include a library name, the design unit is compiled into the specified library.
- If you include a view, the design unit is given the specified view name.

The following example uses the -specificunit option to compile one design unit in a file that contains multiple units.

```
;# Use ncelp to display the hdl.var file. Files with a .v extension in the
;# alu/ directory will be compiled into worklib with a view name of behav.
% ncelp -hdlvar
ncelp: v1.2.(b1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
Parsing -HDLVAR file /usr1/jabbar/hdl.var.
hdl.var files:
1: /usr1/jabbar/hdl.var
Variables defined:
Defined in /usr1/jabbar/hdl.var:
Line # Name      Value
-----  -----
1      NCVLOGOPTS -messages
2      NCELABOPTS -messages
3      NCSIMOPTS -messages
5      LIB_MAP     ( /usr1/jabbar/inca/alu => worklib , + => designlib )
8      VIEW_MAP    ( .v  => behav , .rtl => rtl , .gate => gate , + => module )
```

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with ncvlog

---

```
;# Use -specificunit to compile only the design unit called arith. Module arith
;# is compiled into worklib.arith:behav
% ncvlog -specificunit arith ~/inca/alu/16bit_alu.v
ncvlog: v1.2.(b1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
file: /usr1/jabbar/inca/alu/16bit_alu.v
    module worklib.arith:behav
        errors: 0, warnings: 0
;# Compile only the design unit called arith. Specify the library in the
;# argument to -specificunit. Module arith is compiled into
;# designlib.arith:behav.
% ncvlog -specificunit designlib.arith 16bit_alu.v
ncvlog: v1.2.(b1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
file: 16bit_alu.v
    module designlib.arith:behav
        errors: 0, warnings: 0

;# Compile only the design unit called arith. Compile it into designlib with a
;# view name of rtl.
% ncvlog -specificunit designlib.arith:rtl 16bit_alu.v
ncvlog: v1.2.(b1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
file: 16bit_alu.v
    module designlib.arith:rtl
        errors: 0, warnings: 0
```

## The `worklib and `view Compiler Directives

Using the `worklib or `view compiler directives in your source code overrides any other method of specifying which library you want to compile your design units into or which view name you want them to have.

Syntax:

```
`worklib library_name
`view view_name
```

Examples:

```
`worklib design_lib
`view rtl
```

These compiler directives must be used outside of the module.

Both compiler directives remain in effect for the remaining units in the file until the compiler encounters another `worklib or `view directive, which redefines the library or view, or

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with ncvlog

---

until it encounters a `noworklib or `noview directive, which cancels any previous directives.

In the following example, compiler directives are used to override the library and view mappings specified in the hdl.var file.

```
# hdl.var file
# By default, compile the design units in the file 16bit_alu.v into worklib
# with a view name of behav.

DEFINE WORK worklib
DEFINE VIEW_MAP (.v => behav,\ 
                 .rtl => rtl)

module alu_16 (<port_list>);
...
...
...
endmodule

//Compile module logic into designlib with a view name of behav.
`worklib designlib
module logic (<port_list>);
...
...
...
endmodule

//Compile module arith into worklib with a view name of rtl.
`worklib worklib
`view rtl.
module arith (<port_list>);
...
...
...
endmodule

// Cancel previous `worklib and `view compiler directives. Compile module
// test_alu into worklib with a view name of behav.
`noworklib
`noview
module test_alu;
...
```

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with ncvlog

---

```
...
...
endmodule
;# Compile the design units in 16bit_alu.v.
% ncvlog -messages 16bit_alu.v
ncvlog: v1.2.(b1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
file: 16bit_alu.v
    module worklib.alu_16:behav
        errors: 0, warnings: 0
    module designlib.logic:behav
        errors: 0, warnings: 0
    module worklib.arith:rtl
        errors: 0, warnings: 0
    module worklib.test_alu:behav
        errors: 0, warnings: 0
```

## Mapping of Modules Defined Within `include Files

If a module is defined within a file that was included by another file, the compiler searches for an explicit library and view mapping for the included file. The `worklib/`view compiler directives, the -work/-view command-line options, and the WORK/VIEW variables all provide explicit mappings. Your definition of the LIB\_MAP variable, however, may or may not specify an explicit mapping for the `include file(s).

With the LIB\_MAP variable, if no explicit library mapping is found for a module defined in a `include file , the compiler searches for an explicit mapping for the file that includes the `include file. This search continues recursively until an explicit mapping is found. If no mapping is found, the default case (specified by + => *library*) is used.

The file that is used to select the library mapping is also used to select the view mapping.

The following example illustrates the mapping of modules defined within `include files.

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with ncvlog

---

```
// File top.v
`include "sub1.i"

// File sub1.i
`include "sub2.h"

// File sub2.h
module sub2 ();
  ...
  ...
endmodule

% nchelp -hdlvar
nchelp: v1.2.(b1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
Parsing -HDLVAR file ./hdl.var.

hdl.var files:
1: ./hdl.var

Variables defined:
# No explicit mapping for file sub2.h or for file sub1.i. The explicit mapping
# for file top.v will be used.

# The file that was used to select the library mapping (top.v) is used to
# select the view mapping. Module sub2 will be compiled into
# toplib.sub2:verilog.

Defined in ./hdl.var:
Line # Name      Value
-----  -----
1      LIB_MAP   ( top.v => toplib, + => worklib )
2      VIEW_MAP  ( .v => verilog, .i => include, .h => header )

% ncvlog -messages top.v
ncvlog: v1.2.(b1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
file: top.v
  module toplib.sub2:verilog
    errors: 0, warnings: 0

% ncelp -hdlvar
ncelp: v1.2.(b1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
Parsing -HDLVAR file ./hdl.var.

hdl.var files:
1: ./hdl.var

Variables defined:
```

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with ncvlog

---

```
# No explicit mapping for file sub2.h. The explicit mapping for file sub1.i
# will be used.

# The file that was used to select the library mapping (sub1.i) is used to
# select the view mapping. Module sub2 will be compiled into
# sub1lib.sub2:include.

Defined in ./hdl.var:

Line # Name      Value
-----  ----
1      LIB_MAP   ( top.v => toplib, sub1.i => sub1lib, + => worklib )
2      VIEW_MAP  ( .v => verilog, .i => include, .h => header )

% ncvlog -messages top.v
ncvlog: v1.2.(b1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
file: top.v
      module sub1lib.sub2:include
      errors: 0, warnings: 0

% ncelp -hdlvar
ncelp: v1.2.(b1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
Parsing -HDLVAR file ./hdl.var.
hdl.var files:
1: ./hdl.var
Variables defined:
# The explicit mapping for file sub2.h will be used.

# The file that was used to select the library mapping (sub2.h) is used to
# select the view mapping. Module sub2 will be compiled into
# sub2lib.sub2:header.

Defined in ./hdl.var:

Line # Name      Value
-----  ----
1      LIB_MAP   ( top.v => toplib, sub1.i => sub1lib, \
                     sub2.h => sub2lib, + => worklib )
2      VIEW_MAP  ( .v => verilog, .i => include, .h => header )

% ncvlog -messages top.v
ncvlog: v1.2.(b1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
file: top.v
      module sub2lib.sub2:header
      errors: 0, warnings: 0
```

## Defining Macros on the Command Line

Use the `-define` option to define a macro on the command line. You can:

- Define a variable name as an empty text macro. For example:

```
-define structural
```

Empty text macros are used to define variable names to control conditional compilation. See ["Conditionally Compiling Source Code"](#) on page 163.

- Define a macro name as a string. The `-define` option has the following syntax when it defines a macro name as a string:

```
-define macro_name=macro_string
```

For example, the following option defines the macro name `gate` as the string `or`:

```
-define gate=or
```

Enclose the `macro_string` in quotation marks if it includes characters that you explicitly want as part of the input. For example:

```
-define foo "16'h03"
```

You can define only one macro with each `-define` on the command line, but the number of `-define` options on the command is unlimited. The option cannot define macros of more than one line.

If you define the same macro name differently with a ``define` compiler directive and a command line `-define` option, the command-line option overrides the compiler directive.

When you compile the code in the following example, the ``define` compiler directive defines macro `gate` as an AND gate. You then elaborate the model with `ncelab`. When you simulate with `ncsim`, the value of `c` changes from `x` to 0 to 1.

```
% more def.v
module def();
`define gate and
reg a, b;
`gate (c, a, b);
initial
begin
#1;
a = 0;
b = 1;
$display("a=", a, " b=", b, " c=", c);
#5;
$display("a=", a, " b=", b, " c=", c);
```

## NC-Verilog Simulator Help

### Compiling Verilog Source Files with ncvlog

---

```
a = 1;
#5;
$display("a=", a, " b=", b, " c=", c);
$finish;
end
endmodule
% ncvlog def.v
ncvlog: v1.0.(p1): (c) Copyright 1995, 1996 Cadence Design Systems, Inc.
% ncelab def
ncelab: v1.0.(p1): (c) Copyright 1995, 1996 Cadence Design Systems, Inc.
% ncsim def
ncsim: v1.0.(p1): (c) Copyright 1995, 1996 Cadence Design Systems, Inc.
ncsim> run
a=x    b=x    c=x
a=0    b=1    c=0
a=1    b=1    c=1
Simulation complete via $finish(1) at time 11 NS + 0
./def.v:17 $finish;
ncsim> exit

:# The gate is simulated using the command-line definition.
% ncvlog -define gate=or def.v
ncvlog: v1.0.(p1): (c) Copyright 1995, 1996 Cadence Design Systems, Inc.
`define gate and
      |
ncvlog: *W,MACNDF (def.v,2|16): text macro 'gate' not redefined
      using command line definition.
% ncelab def
ncelab: v1.0.(p1): (c) Copyright 1995, 1996 Cadence Design Systems, Inc.
% ncsim def
ncsim: v1.0.(p1): (c) Copyright 1995, 1996 Cadence Design Systems, Inc.
ncsim> run
a=x    b=x    c=x
a=0    b=1    c=1
a=1    b=1    c=1
Simulation complete via $finish(1) at time 11 NS + 0
./test.v:16 $finish;
ncsim> exit
```

---

## Elaborating the Design with ncelab

---

This chapter contains the following sections:

- [Overview](#)
- [ncelab Command Syntax](#)
- [ncelab Command Options](#)
- [Example ncelab Command Lines](#)
- [hdl.var Variables](#)

Additional topics:

- [How Modules and UDPs Are Resolved During Elaboration](#)
- [Enabling Read, Write, or Connectivity Access to Simulation Objects](#)
- [Disabling Timing in Selected Portions of a Design](#)
- [Selecting a Delay Mode](#)
- [Setting Pulse Controls](#)

## Overview

Before you can simulate your model, the design hierarchy defining the model must be elaborated. The tool you use for elaborating the design is called *ncelab*.

*ncelab* is a language-independent elaborator. It constructs a design hierarchy based on the instantiation and configuration information in the design, establishes signal connectivity, and computes initial values for all objects in the design. The elaborated design hierarchy is stored in a simulation snapshot, which is the representation of your design that the simulator uses to run the simulation. The snapshot is stored in the library database file along with the other intermediate objects generated by the compiler and elaborator.

If you are running the NC-Verilog simulator using the single-step invocation method (*ncverilog*), and want to compile your source files and elaborate the design, use the `+elaborate` option. This option stops the simulator after elaboration.

If you are running the NC-Verilog simulator using multi-step invocation, invoke *ncelab* with command-line options and the `Library.Cell[:View]` name(s) of the top-level HDL design unit(s). You can specify the arguments in any order, but parameters to options must immediately follow the options they modify.

The top-level unit(s) specified on the command line can be:

- One VHDL top-level unit.
- One or more Verilog top-level units.
- One VHDL unit and one or more Verilog units.

Design units specified on the command line cannot be instantiated in the design.

Syntax:

```
% ncelab [options] [Lib.]Cell[:View] ...
```

- You must specify the cell (top-level unit name).
- You must specify the library if a top-level unit with the same name exists in more than one library.
- If there are multiple views of the top-level unit(s), the easiest (and recommended) thing to do is to specify the view on the command line. If you do not specify the view, *ncelab* uses the following rules to resolve the reference to the top-level design unit:

- a. Search the library defined with the `WORK` variable in the `hdl.var` file. If one view of the cell exists in that library, use that view. Generate an error message if more than one view exists.
- b. If the `WORK` variable is not defined in the `hdl.var` file, search the libraries defined in the `cds.lib` file. If one view of the cell exists in the libraries, use that view. Generate an error message if more than one view exists.

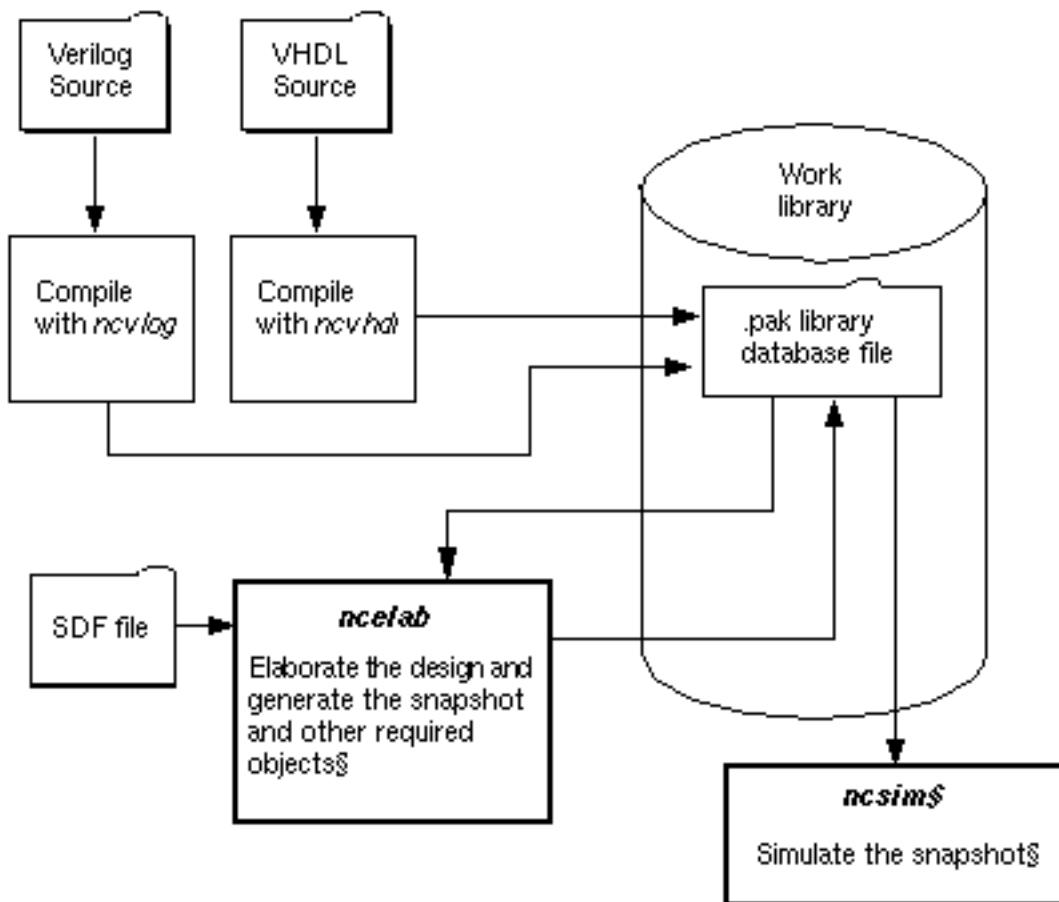
Elaboration produces a simulation snapshot. The snapshot is also a `Lib.Cell:View`. Assuming that the `-snapshot` option was not used to explicitly name the snapshot, the snapshot is named:

- *Library*—the name of the library where the top-level unit on the `ncelab` command line was found. If more than one Verilog top-level module is specified on the command line, the *Library* is the name of the library where the first top-level module listed on the command line was found.
- *Cell*—the name of the top-level unit on the `ncelab` command line. If more than one Verilog top-level module is specified on the command line, the *Cell* is the name of the first top-level module listed on the command line.
- *View*—the view name that was specified for the first top-level design unit on the `ncelab` command line or (if a view was not specified) the name of the view that was used as a result of the rules that `ncelab` uses to resolve references to top-level units given on the `ncelab` command line.

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

The following figure illustrates the *ncelab* process.



By default, the elaborator marks all simulation objects in the design as having no read, write, or connectivity (load and driver) access. Turning off these three forms of access allows the elaborator to perform a set of optimizations that can dramatically improve simulation performance. However, turning off access to the HDL data structures means that you will not be able to access these objects from a point outside the HDL code through Tcl commands or through PLI, VPI/VHPI. You can set the global visibility access to simulation objects with the [-access](#) option when you invoke *ncelab*. You can also use the [-afile](#) option to include an access file, which lets you set the visibility access for particular instances or portions of a design.

**Note:** If you are running the NC-Verilog simulator with the `ncverilog` command, use the `+ncaccess+` option to turn on specific kinds of access to all objects in the design. Use `+ncafile+` to include an access file. See [Chapter 4, “Running NC-Verilog with the ncverilog Command,”](#) for details on `ncverilog`.

When you change any of the design units in the hierarchy, you must recompile the design units that you have changed and re-elaborate the design hierarchy. You can automatically recompile all out-of-date design units and re-elaborate the design by:

- Running *ncupdate*. This utility runs *ncvlog* to recompile any changed Verilog design units and *ncvhdl* to recompile any changed VHDL units, and then runs *ncelab* to re-elaborate the design. *ncelab* also automatically invokes the *ncsdfc* utility to recompile the SDF source file if it detects a change in the file. The elaborator then generates a new snapshot. Use *ncupdate* when you want to update the snapshot, but do not want to simulate. See “[ncupdate](#)” on page 928 for information on *ncupdate*.
- Including the *-update* option on the *ncsim* command. This option calls *ncupdate*, which recompiles any changed design units, recompiles the SDF file if necessary, re-elaborates the design, generates a new snapshot, and then invokes the simulator. Use *ncsim -update* if you want to update and simulate. See “[Updating Design Changes When You Invoke the Simulator](#)” on page 332.

You must elaborate the entire design at least once before you can use either feature to automatically update the design.

You can also run *ncelab* to automatically generate a VHDL configuration file by including the *-conffile* option. The syntax for the *ncelab* command is as follows:

```
ncelab [options] -conffile configuration_filename [lib.]cell[:view]
```

See the description of the *-conffile* option for details on generating a configuration file.

## ncelab Command Syntax

The *ncelab* command-line options listed in this section are divided into three groups:

- General options, which apply to both languages.
- VHDL only options, which apply only to the VHDL portions of a design.
- Verilog only options, which apply only to the Verilog portions of a design.

Options can be abbreviated to the shortest unique string, indicated here with capital letters.

*ncelab [options] [Lib.]Cell[:View] ...*

### General Options

```
[-ACcess [+/-] access_spec]
[-AFile access_file]
[-APPend log]
[-CDslib cdslib_pathname]
[-COverage]
[-ERrormax integer]
[-EXPand]
[-File arguments_filename]
[-GENAfile access_filename]
[-HDLvar hdlvar_pathname]
[-HELP]
[-INtermod path]
[-LOGfile filename]
[-MAXdelays]
[-MESSages]
[-MIndelays]
[-NCError warning_code]
[-NCFatal {warning_code | error_code}]
[-NEverwarn]
[-NOCopyright]
[-NODEadcode]
[-NOLog]
[-NOSOource]
[-NOSTdout]
[-NO_TCHK_Msq]
[-NOTImingchecks]
[-NOWarn warning_code]
```

```
[-NTc warn]
[-OMICheckinglevel checking_level]
[-SDF Cmd file sdf_command_file]
[-SDF NO Warnings]
[-SDF Precision precision]
[-SDF Verbose]
[-SNAPSHOT snapshot_name]
[-STatus]
[-TYPedelays]
[-UPdate]
[-VErsion]
```

## VHDL Only Options

```
[-CONFFile configuration_filename]
[-COMpile]
[-CONFFLat]
[-CONFName configuration_name]
[-Overwrite]
[-PROmpt]
[-USEArch priority_list_of_architectures]
[-GENeric generic_name => value]
[-NOIPd]
[-NOVitalaccl]
[-NOTCHK_Xgen]
[-NO_VPD_Msg]
[-NO_VPD_Xgen]
[-PReserve]
[-Relax]
[-USe5x4vhdl]
[-V93]
[-VIPDMAX]
[-VIPDMIn]
[-Work work_library]
```

## Verilog Only Options

```
[-ANno simtime]
[-Binding [lib.]cell[:view]]
[-DElay mode {zero | unit | path | distributed | none}]
[-DIable enht]
```

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

```
[-EPULSE_NEq]
[-EPULSE_NOneq]
[-EPULSE_ONDetect]
[-EPULSE_ONEvent]
[-EXTEND_TCHECK Data limit percent_relaxation]
[-EXTEND_TCHECK Reference limit percent_relaxation]
[-IEee1364]
[-LIBVerbose]
[-LOADPli1 shared_lib_name:boot_func_name[,boot_func_name ...]]
[-LOADVpi shared_lib_name:boot_func_name[,boot_func_name ...]]
[-NEG_tchk]
[-NOAutosdf]
[-NONEq_tchk]
[-NONNotifier]
[-NO_Sdfa_header]
[-PAthpulse]
[-PLINOOptwarn]
[-PLINOWarn]
[-PULSE_E error_percent]
[-PULSE_INT_E error_percent]
[-PULSE_INT_R reject_percent]
[-PULSE_R reject_percent]
[-SDF_NOCHECK_Celltype]
[-SDF_Worstcase_rounding]
[-TFile timing_file]
[-TImescale 'time_unit / time_precision' ]
```

## ncelab Command Options

This section describes the options that you can use with the `ncelab` command. Options can be entered in upper or lower case. Capital letters indicate the shortest possible abbreviation for an option.

The options listed in this section apply to both languages unless specifically noted.

### **-ACcess [+/-] *access\_spec***

Set the visibility access for all objects in the design. The *access\_spec* argument can be:

- `r` (read access)
- `w` (write access)
- `c` (connectivity access)
- Any combination of these three access types

Use the plus sign ( `+` ) to turn on the specified access. Use the minus ( `-` ) sign to turn off the specified access. If no plus or minus sign is used, `+` is the default. The `+` and `-` options apply to all subsequent `r`, `w`, or `c` specifications until the next `+` or `-`.

By default, objects do not have read, write, or connectivity access. In other words, the default is `-access -r-w-c`.

Objects that are given write access are also given read access. Objects that are given connectivity access are also given write access, and, therefore, read access.

Examples:

1. Read access only

```
-access +r (same as -access r)
```

2. Write access (objects also get read access)

```
-access +w (same as -access w)
```

3. Read/Write access

```
-access +r+w (same as -access +rw or -access rw)
```

4. Read/Write/Connectivity access

```
-access +r+w+c (same as -access +rwc or -access rwc)
```

## 5. Connectivity access (objects also get read access)

```
-access +c (same as -access c)
```

**Note:** Objects that are given connectivity access are also given write and read access. The following option results in connectivity, write, and read access to all objects:

```
-access +c-rw
```

You can also use multiple -access options. For example:

```
-access +r -access -w
```

See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for more information.

### **-AFile access\_file**

Use the specified access file. An access file is a text file that lets you set the visibility access for particular instances or portions of a design. See “[Using -afile to Include an Access File](#)” on page 253 for details on writing and using an access file.

Use the `+ncafile+access_file` option if you are running *ncverilog*.

Use the [-access](#) option to specify global visibility access for all objects in the design.

### **-ANno\_simtime**

#### **(Verilog only)**

Enable the use of PLI/VPI routines that modify delays at simulation time. These routines are `acc_replace_delays`, `acc_append_delays`, and `vpi_put_delays`.

If this option is not specified at elaboration time, and a PLI/VPI routine that modifies delays is executed at simulation time, a message is issued and the delay modification does not take place.

This option disables optimizations in the simulator that take delays into account, and will, therefore, have some performance impact. Use this option only if you intend to modify delays at simulation time.

Using this option sets the default access to simulation objects to read/write when the design is elaborated. Do not use this option if you want to run in regression mode.

**Note:** Negative limit values in `$setuphold` or `$recrem` timing checks cannot be modified using PLI/VPI routines.

### **-APPEND\_LOG**

Append log information from multiple runs of *ncelab* to one log file. Use this option if you are going to run *ncelab* multiple times and you want all log information appended to one log file. If you do not use this option, the log file is overwritten each time you run *ncelab*.

If you use both `-append_log` and `-nolog` on the command line, `-nolog` overrides `-append_log`.

Because the log file is opened before variables in the `hdl.var` file are read, the `-append_log` option is ignored with a warning if you define it with the `NCELABOPTS` variable in an `hdl.var` file.

### **-BINDING [lib.]cell[:view]**

Force an explicit submodule binding. See “[How Modules and UDPs Are Resolved During Elaboration](#)” on page 233 for more information.

### **-CDSLIB cdslib\_pathname**

Use the specified `cds.lib` file. See “[The cds.lib File](#)” on page 110 for details on the `cds.lib` file.

All tools and utilities that require a `cds.lib` file use a default search mechanism to find the `cds.lib` file. See “[The setup.loc File](#)” on page 131 for information on this search mechanism. Use the `-cdslib` option to override the default search order and force the elaborator to use the specified `cds.lib` file.

Example:

```
% ncelab -cdslib ~/design_lib/cds.lib top
```

The elaborator reads the `cds.lib` file before it processes any variables defined in the `hdl.var` file. You cannot, therefore, include the `-cdslib` option with the `NCELABOPTS` variable in an `hdl.var` file.

## **-COMpile**

### **(VHDL only)**

Compile the configuration file after creating it.

You can use the `-compile` option only when you are generating a VHDL configuration file with the `-conffile` option.

## **-CONFFile *configuration\_filename***

### **(VHDL only)**

Generate a VHDL configuration file with the specified name for the design unit specified on the command line. When you include the `-conffile` option to generate a configuration file, the elaborator generates the configuration file and then exits. The design is not actually elaborated.

To generate a configuration file, use the following command:

```
ncelab [options] -conffile configuration_filename [lib.]cell[:view]
```

The [*lib.*]*cell[:view]* argument is the name of a compiled VHDL design unit.

The *configuration\_filename* argument to the `-conffile` option is the filename of the configuration file. This argument is required.

By default, the elaborator generates a hierarchical configuration file that contains a separate configuration specification for each entity. In this case, the name of the output configuration file is *library\_filename*. For example, the following command generates a configuration in a file called *worklib\_abc.vhd*.

```
% ncelab -conffile abc.vhd WORKLIB.TEST_DFF:TEST_BENCH_DFF
```

Use the `-confflat` option if you want to generate a single flat configuration specification for the entire design. The name of the output configuration file is the name that you specify on the command line. For example, the following command generates a flat configuration in a file called *abc.vhd*.

```
% ncelab -conffile abc.vhd WORKLIB.TEST_DFF:TEST_BENCH_DFF -confflat
```

**Note:** If your VHDL design has entities that are compiled into different libraries, the elaborator generates multiple configuration files in the current directory for the default hierarchical format so that the configuration for an entity can be compiled in the entity's library. These configuration files are called *libraryname\_filename* (for example, *lib1\_cfg.vhd*,

`lib2_cfg.vhd`, and `lib3_cfg.vhd`). For the flat configuration file format, the elaborator always generates a single file.

If an explicit binding is not specified in the VHDL source code, the elaborator, by default, selects the most-recently analyzed architecture for an entity. You can override this selection by including the `-prompt` or the `-usearch` option on the command line.

- `-prompt`—Displays a list of all architectures that are available for an entity and asks the user to select one.
- `-usearch priority_list_of_architectures`—Uses the specified priority list of architectures when selecting an architecture for an entity. The `priority_list_of_architectures` argument can be a single architecture or a list of architectures. If you specify more than one architecture, separate the items on the list with a comma. No spaces are allowed in the list.

The following table summarizes the selection of an architecture for an entity when writing a configuration file:

---

<b>-prompt option?</b>	<b>Priority list of architectures (-usearch option)?</b>	<b>Action</b>
No	No	The elaborator selects the most-recently analyzed architecture for each component's entity.
Yes	No	If an explicit binding is not found, the elaborator displays a list of available architectures for an entity, and prompts you to select one of them.
No	Yes	<p>If an explicit binding is not found, the elaborator considers (in order) the architectures on the priority list and uses the first one that it finds.</p> <p>If the priority list is exhausted, the elaborator selects the most-recently analyzed architecture.</p>

<b>-prompt option?</b>	<b>Priority list of architectures (-usearch option)?</b>	<b>Action</b>
Yes	Yes	<p>If an explicit binding is not found, the elaborator considers (in order) the architectures on the priority list and uses the first one that it finds.</p> <p>If the priority list is exhausted, the elaborator displays a list of available architectures for an entity, and prompts you to select one of them.</p>

Other command-line options that can be used only with the `-conffile` option are:

- `-compile`—Compiles the configuration file after generating it.
- `-confname configuration_name`—Uses the specified name for the configuration instead of the default name, which is `cfg_entity_architecture`.
- `-overwrite`—Overwrites an existing configuration file that has the same name.

Most elaborator options do not apply when you are generating a configuration file with the `-conffile` option because no elaboration of the design actually takes place. You can use the following elaborator command-line options when you are using *ncelab* to generate a configuration file:

<code>-append_log</code>	<code>-neverwarn</code>
<code>-cdslib cdslib_pathname</code>	<code>-nocopyright</code>
<code>-errormax integer</code>	<code>-nolog</code>
<code>-file arguments_filename</code>	<code>-nostdout</code>
<code>-hdlvar hdl_pathname</code>	<code>-v93</code>
<code>-logfile filename</code>	<code>-work work_library</code>
<code>-messages</code>	

See “[Using ncelab to Generate a VHDL Configuration File](#)” on page 228 for examples of generating a configuration file.

## **-CONFFlat**

### **(VHDL only)**

Generate a VHDL flat configuration declaration.

By default, the elaborator generates a hierarchical configuration file that contains a separate configuration specification for each component. Use the `-confflat` option if you want to generate a single configuration specification for the entire design.

You can use the `-confflat` option only when you are generating a VHDL configuration file with the `-conffile` option.

See “[Using ncelab to Generate a VHDL Configuration File](#)” on page 228 for examples of a hierarchical and a flat configuration file.

## **-CONFName *configuration\_name***

### **(VHDL only)**

Use the specified name for the configuration instead of the default name. By default, the name of the configuration is `cfg_entity_architecture`.

For a hierarchical configuration, in which a separate configuration specification is generated for each entity, the configuration name that you specify with the `-confname` option is used for the top-level unit.

You can use the `-confname` option only when you are generating a VHDL configuration file with the `-conffile` option.

## **-COverage**

Enable code coverage for the entire design.

**Note:** Code coverage is not currently supported on Windows platforms.

If you are running the NC-Verilog simulator using single-step invocation, use the `+nccoverage` option as follows:

```
% ncverilog +nccoverage Verilog-XL_command_line
```

Three types of code coverage are available:

- Statement coverage, which tells you whether a particular statement was executed or was not executed during a given simulation run. The code coverage analyzer reports the statement coverage statistic in boolean format, which reports 1 if the statement has executed and 0 if the statement has not executed.
- State machine (fsm) coverage, which tells you which states the state machine variables have been in (state visitation), and which state-to-state transitions have taken place (state transitions).

To enable state machine coverage, you must use the `ncelab -access +r` option (`ncverilog +ncaccess+r`) to provide read access to Verilog objects.

- Expression coverage, which identifies which terms of an expression in your code contribute to true and false values for the expression's terms during a simulation run.

To enable expression coverage, you must compile the Verilog modules (and VHDL design units) with the `-linedebug` option.

**Note:** Using the expression recording features can severely impact your simulation performance.

For example, the following command line enables statement coverage for the entire design:

```
% ncelab work.tb_full_add -coverage
```

After you enable code coverage with the `-coverage` option, invoke the simulator and then use the Tcl coverage command to specify the kind of code coverage that you want and to control code coverage dumping. If you are using the SimVision analysis environment, invoke the simulator with the `-gui` option and then use the commands on the *Coverage* menu. See the *Code Coverage User Guide* for details on code coverage and on the `coverage` command. See “[Gathering Code Coverage Data](#)” in the *SimVision Analysis Environment User Guide* for details on using the *Coverage* menu commands.

Code coverage for mixed-language designs is supported in the Cadence® NC-Sim mixed language simulator.

**-DElay\_mode {zero | unit | path | distributed | none}**

**(Verilog only)**

Use the specified delay mode for the Verilog portions of the hierarchy. The argument can be: zero, unit, path, distributed or none.

See “[Selecting a Delay Mode](#)” on page 270 for more information on specifying a delay mode.

**-Disable\_enht**

**(Verilog only)**

Disable enhanced timing features. These timing features are enabled by using special properties in a specify block. Using the properties gives you more control over the selection of a delay when there are multiple inputs that occur either simultaneously or while a path delay output is already scheduled. See “[Specify Properties for Module Path Delays](#)” on page 776 for more information.

**-EPULSE\_NEg**

**(Verilog only)**

Filter cancelled events (negative pulses) to the e state. This option makes cancelled events visible. Using this option overrides any showcancelled and noshowcancelled settings in specify blocks. See “[Pulse Filtering and Cancelled Schedules](#)” on page 288 for more information.

**-EPULSE\_NOneg**

**(Verilog only)**

Do not filter cancelled events (negative pulses) to the e state. Using this option overrides any showcancelled and noshowcancelled settings in specify blocks. See “[Pulse Filtering and Cancelled Schedules](#)” on page 288 for more information.

**-EPULSE\_ONDetect**

**(Verilog only)**

Use On-Detect filtering of error pulses. This option extends the e state back to the edge of the event that caused the pulse to occur.

See “[Pulse Filtering Style](#)” on page 285 for details on On-Detect and On-Event pulse filtering styles.

## **-EPULSE\_ONEvent**

**(Verilog only)**

Use On-Event filtering of error pulses.

See “[Pulse Filtering Style](#)” on page 285 for details on On-Detect and On-Event pulse filtering styles.

## **-ERrormax *integer***

Abort after reaching the specified number of errors. By default, there is no limit on the number of error messages.

By using `-errormax`, you can limit the number of errors that are generated, fix those errors, and then rerun to check for other errors. This option is useful when you are running a large design that might contain numerous errors.

Example:

```
% ncelab -errormax 10 top
```

## **-EXPand**

Expand all vector nets.

For NC-Verilog, this option expands vector nets the same way that the `-x` option in Verilog-XL expands vector nets. Using `-expand` will thus eliminate mismatches due to how vectored nets are expanded when you compare databases generated by the two simulators.

Using the `-expand` option is required in order to set a breakpoint on a subelement of a vector Verilog wire or VHDL signal. By default, all vectors are compressed, and a `stop -object` command can only be used on the entire object.

**Note:** Using this option can severely impact performance.

## **-EXTEND\_TCHECK\_Data\_limit *percent relaxation***

**(Verilog only)**

Extend the violation regions established by a pair of setuphold or recrem timing checks with negative values in which the timing checks contain two different constraints for posedge and

negedge of data with respect to the same reference signal and in which the violation regions do not overlap.

In situations where there are two setuphold or recrem timing checks that establish two different constraints for posedge and negedge of data with respect to the same reference signal, violation regions may not overlap. Because the violation regions created by the timing checks do not overlap, 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.

You can avoid this non-convergence by hand-editing the timing checks in the HDL or in the SDF file to create some overlap, or you can use the `-extend_tcheck_data_limit` or the `-extend_tcheck_reference_limit` option to automatically extend the violation regions by the specified percentage to create the overlap.

The `-extend_tcheck_data_limit` option changes the hold or recovery limit in the timing checks so that the violation regions overlap by at least two units of simulation precision. The `percent_relaxation` argument is the maximum percentage increase allowed in the timing violation window to achieve the overlap.

You cannot use both `-extend_tcheck_data_limit` and `-extend_tcheck_reference_limit` on the command line. Using these options automatically turns on the `-ntc_warn` option.

When you use either of these options, the elaborator issues a warning message (NTCRLX) to let you know 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.

Example:

```
% ncelab -extend_tcheck_data_limit 100 worklib.test:module
```

See “[Non-Convergence in Timing Checks](#)” on page 734 for more information.

### **`-EXTEND_TCHECK_Reference_limit percent_relaxation`**

**(Verilog only)**

Extend the violation regions established by a pair of setuphold or recrem timing checks with negative values in which the timing checks contain two different constraints for posedge and negedge of data with respect to the same reference signal and in which the violation regions do not overlap.

In situations where there are two setuphold or recrem timing checks that establish two different constraints for posedge and negedge of data with respect to the same reference signal, violation regions may not overlap. Because the violation regions created by the timing checks do not overlap, 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.

You can avoid this non-convergence by hand-editing the timing checks in the HDL or in the SDF file to create some overlap, or you can use the `-extend_tcheck_data_limit` or the `-extend_tcheck_reference_limit` option to automatically extend the violation regions by the specified percentage to create the overlap.

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 `percent_relaxation` argument is the maximum percentage increase allowed in the timing violation window to achieve the overlap.

You cannot use both `-extend_tcheck_data_limit` and `-extend_tcheck_reference_limit` on the command line. Using these options automatically turns on the `-ntc_warn` option.

When you use either of these options, the elaborator issues a warning message (NTCRLX) to let you know 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.

Example:

```
% ncelab -extend_tcheck_reference_limit 100 worklib.test:module
```

See “[Non-Convergence in Timing Checks](#)” on page 734 for more information.

### ***-File arguments\_filename***

Use the command-line arguments contained in the specified arguments file.

You can store frequently used or lengthy command lines by putting command-line arguments (command options and top-level design unit names) in a text file. When you invoke the elaborator with the `-file` option, the arguments in the arguments file are incorporated with your command as if they had been entered on the command line.

The arguments file can contain command options, including other `-file` options, and top-level design unit names. The individual arguments within the arguments file must be separated by white space or comments.

Example:

```
% ncelab -file ncelab.args top
```

You can also use the `NCELABOPTS` variable in an `hdl.var` file to include command-line options and top-level design unit names.

### **-GENAfile *access\_filename***

Generate an access file that has the specified filename.

Use the `+ncgenafile+access_filename` option if you are running *ncverilog*.

This option creates an access file based on the objects that were accessed during simulation by Tcl commands or by a PLI application and on the type of access that was required. You can then use this access file in a subsequent run by including it with the [-afile](#) option (or `+ncafile+` if you are running *ncverilog*). See “[Generating an Access File](#)” on page 260 for more information.

See “[Using -afile to Include an Access File](#)” on page 253 for information on the access file.

### **-GENERIC *generic\_name => value***

**(VHDL only)**

Specifies a value for a top-level generic.

This option associates a value with a top-level generic on the command line. The `generic_name` is the name of the generic as it appears in the VHDL source (case is ignored). The `value` is an appropriate value for the declared data type of the generic.

The generic subtypes are limited to:

```
generic (G: INTEGER := ...)  
generic (G: NATURAL := ...)  
generic (G: POSITIVE := ...);  
generic (G: TIME := ...)  
generic (G: STRING := ...)  
generic (G: BOOLEAN := ...)
```

These subtypes are declared in `STD.STANDARD`. The default value expression is optional. The shown type marks must be used; no constraint is allowed (the constraint for generics of type `STRING` is taken from the associated value).

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

Integers must be decimal literals (not based literals). The literals can contain underscores, but the restrictions on leading, trailing, and double underscores are not enforced.

Time literals must be decimal physical literals. The abstract literal portion (number) must be present; the preceding comments for integers apply to TIME literals. The unit portion of the physical literal is required and can be any time unit name from fs through hr.

Strings must be delineated by double quotes. Colons are not allowed as alternate delimiters. You can embed double quotes in the string as you normally do.

Given the following entity and architecture declarations:

```
entity E is
    generic (G1: INTEGER:= 0; G2: TIME:= 0 ns; G3: STRING:="" ; G4: BOOLEAN:=FALSE;
             G5: NATURAL:= 10; G6: POSITIVE:= 5);
end;

architecture A of E is
begin
    ...
end;
```

The following ncelab commands are legal:

```
ncelab -generic "G1=>99" e:a
ncelab -generic 'g1 => -99' e:a
ncelab -generic "G2 => 17 ns" e:a
ncelab -generic 'G3 => "abc"' e:a
ncelab -generic 'G3 => "a""c"' e:a
ncelab -generic "G1=>-99" -generic "G2=>1us" -generic G3=\>\xyzzy\ e:a
ncelab -generic 'G4 => "TRUE"' e:a
```

The following ncelab commands are not legal:

```
ncelab -generic "G1 => 16#FF#" e:a      -- no based literals
ncelab -generic G2 => 0.1 ns" e:a        -- no real literals; use 100 ps
ncelab -generic "G3 => abc"              -- no quotes around abc
ncelab -generic "G4 => 0"                 -- value can only be "true" or "false"
```

#### **-HDLvar *hdlvar\_pathname***

Use the specified `hdl.var` file. See “[The hdl.var File](#)” on page 118 for details on the `hdl.var` file.

All tools and utilities that require an `hdl.var` file use a default search mechanism to find the `hdl.var` file. See “[The setup.loc File](#)” on page 131 for information on this search mechanism. Use the `-hdlvar` option to override the default search order and force the elaborator to use the specified `hdl.var` file.

**Example:**

```
% ncelab -hdlvar ~/hdl.var alu_16
```

You cannot include the `-hdlvar` option with the `NCELABOPTS` variable in an `hdl.var` file.

## **-HElp**

Display a list of the `ncelab` command options with a brief description of each option.

```
% ncelab -help
```

This option is ignored if you include it with the `NCELABOPTS` variable in an `hdl.var` file.

## **-IEee1364**

**(Verilog only)**

Check for compatibility with the IEEE1364 standard.

Use the `-ieee1364` option to check for compatibility with the *IEEE-1364 Verilog Hardware Description Language Reference Manual*. Messages generated by the elaborator contain references to relevant sections of the IEEE-1364 LRM.

Using this option is important if you are going to use other tools, such as a second simulator or a synthesis tool, that are compatible only with a particular standard or specification.

Most compatibility checks are performed during compilation. The `-ieee1364` option should be used when you invoke `ncvlog` to compile your Verilog source files.

**Example:**

```
% ncelab -ieee1364 top_mod
```

## **-INtermod\_path**

For Verilog, enable transport delay behavior with pulse control and the ability to specify unique delays for each source-load path. See “[Interconnect Delays](#)” on page 748 for details on interconnect delays.

For VHDL, enable the ability to specify unique delays for each source-load path during VITAL SDF annotation. See “[VITAL SDF Annotation](#)” on page 790 for details on VITAL SDF annotation.

You cannot use the `-intermod_path` option with the `-vipdmax` or `-vipdmin` options.

**-LIBVerbose**

**(Verilog only)**

Display messages during module/udp instantiation.

**-LOADPLI1 *shared\_lib\_name:boot\_func\_name[,boot\_func\_name ...]***

**(Verilog only)**

Dynamically load the specified PLI1.0 application.

If a PLI application has already been compiled into a dynamic shared library, you can use `-loadpli1` to load the library and to register the system tasks defined in the application at run time. If you are using single-step invocation with the `ncverilog` command, use the `+loadpli1` command-line option.

The argument to this option is the name or full path of the shared library that contains the PLI application followed by the name of the function that registers the new system tasks. This function, called the *bootstrap* function, is part of the PLI application, and is defined in the shared library.

You can load any number of applications in the same statement by separating the names of the bootstrap functions with a comma. No spaces are allowed in the argument.

The file extension of the shared library is optional. The elaborator appends a suffix that is consistent with the OS that you are running. For example, if you are running on the SUN4v platform and enter the following command, the elaborator searches for a library called `SSI.so`.

```
% ncelab -loadpli1 SSI:ssi_boot top
```

For PLI1.0 applications, the simulator always loads the shared library and executes any bootstrap function(s) that is passed to the elaborator.

Examples:

1. % ncelab -loadpli1 mylib:boot1

The elaborator loads the shared library and executes boot1. The simulator loads the shared library and executes boot1.

If you are using single-step invocation with the ncverilog command, use the +loadpli1 command-line option.

```
% ncverilog +loadpli1=mylib:boot1
```

2. % ncelab -loadpli1 mylib:boot1,boot2

The elaborator loads the shared library and executes boot1 and boot2. The simulator loads the shared library and executes boot1 and boot2.

For single-step invocation, use the following command:

```
% ncverilog +loadpli1=mylib:boot1,boot2
```

In some cases, you may want to execute different functions in the elaborator and in the simulator. For example, you might have a PLI application that you want to run at simulation time only. Such an application may perform tasks such as recording how long a simulation runs or opening communication with another tool. You can do this by using a period to separate the function that you want to execute in the elaborator from the function that you want to execute in the simulator. For example:

3. % ncelab -loadpli1 mylib:boot1.boot2

```
% ncverilog +loadpli1=mylib:boot1.boot2
```

In this example, the argument contains one *boot\_func\_name*, which is divided into two parts. The elaborator loads mylib and executes boot1. The simulator loads mylib and executes boot2.

**Note:** boot2 must register the same system tasks that are registered by boot1.

4. % ncelab -loadpli1 mylib:boot1,boot2.boot3,boot4

```
% ncverilog +loadpli1=mylib:boot1,boot2.boot3,boot4
```

In this example, the argument contains three boot functions. The second is divided into two parts. The elaborator loads mylib and executes boot1, boot2, and boot4.

The simulator loads mylib and executes boot1, boot3, and boot4.

**-LOADVpi *shared\_lib\_name:boot\_func\_name[,boot\_func\_name ...]***

**(Verilog only)**

Dynamically load the specified VPI application.

If a VPI application has already been compiled into a dynamic shared library, you can use `-loadvpi` to load the library and to register the system tasks and VPI callbacks defined in the application at run time. If you are using single-step invocation with the `ncverilog` command, use the `+loadvpi` command-line option.

The argument to this option is the name or full path of the shared library that contains the VPI application followed by the name of the function that returns a pointer to either a `vpi_register_systf()` or a `vpi_register_cb()` function call that contains the definitions of system tasks and functions. This function, called the *bootstrap* function, is part of the VPI application, and is defined in the shared library.

You can load any number of applications in the same statement by separating the names of the bootstrap functions with a comma. No spaces are allowed in the argument.

The file extension of the shared library is optional. The elaborator appends a suffix that is consistent with the OS that you are running. For example, if you are running on the SUN4v platform and enter the following command, the elaborator searches for a library called `SSI.so`.

```
% ncelab -loadvpi SSI:ssi_boot top
```

Examples:

1. % ncelab -loadvpi mylib:boot1

The elaborator loads the shared library and executes `boot1`. The simulator loads the shared library and executes `boot1`.

If you are using single-step invocation with the `ncverilog` command, use the `+loadvpi` command-line option.

```
% ncverilog +loadvpi=mylib:boot1
```

2. % ncelab -loadvpi mylib:boot1,boot2

The elaborator loads the shared library and executes `boot1` and `boot2`. The simulator loads the shared library and executes `boot1` and `boot2`.

For single-step invocation, use the following command:

```
% ncverilog +loadvpi=mylib:boot1,boot2
```

In some cases, you may want to execute different functions in the elaborator and in the simulator. For example, you might have a PLI application that you want to run at simulation time only. Such an application may perform tasks such as recording how long a simulation runs or opening communication with another tool. You can do this by using a period to separate the function that you want to execute in the elaborator from the function that you want to execute in the simulator. For example:

3. % ncelab -loadvpi mylib:boot1.boot2  
% ncverilog +loadvpi=mylib:boot1.boot2

In this example, the argument contains one *boot\_func\_name*, which is divided into two parts. The elaborator loads `mylib` and executes `boot1`. The simulator loads `mylib` and executes `boot2`.

**Note:** `boot2` must register the same system tasks that are registered by `boot1`.

4. % ncelab -loadvpi mylib:boot1,boot2.boot3,boot4  
% ncverilog +loadvpi=mylib:boot1,boot2.boot3,boot4

In this example, the argument contains three *boot\_func\_names*. The second is divided into two parts. The elaborator loads `mylib` and executes `boot1`, `boot2`, and `boot4`.

The simulator loads `mylib` and executes `boot1`, `boot3`, and `boot4`.

If your VPI application does not contain user-defined system tasks or functions and you use other VPI callbacks (that is, `vpi_register_cb` instead of `vpi_register_systf`), the code only needs to be run at simulation time. In this case, you can use the `-loadvpi` option on the `ncsim` command line as follows:

```
% ncsim -loadvpi mylib:boot1  
% ncverilog +loadvpi=mylib:.boot1
```

### **-LOGfile *filename***

Use the specified name for the log file instead of the default name `ncelab.log`.

Example:

```
% ncelab -logfile mylog.log top
```

Use `-nolog` if you don't want a log file. If you use both `-logfile` and `-nolog` on the command line, `-logfile` overrides `-nolog`.

Use [-append log](#) if you are going to run *ncelab* multiple times and you want all log information appended to one log file. If you do not use this option, the log file is overwritten each time you run *ncelab*.

Because the log file is opened before variables in the *hdl.var* file are read, the *logfile* option is ignored with a warning if you define it with the *NCELABOPTS* variable in an *hdl.var* file.

Redirecting the output of an NC tool to the same log file that the tool creates can result in corrupted information. For example, with the following command, *ncelab* generates *ncelab.log* and then the output is redirected to the same file.

```
% ncelab -messages worklib.top:arch > ncelab.log
```

Instead of using redirection, use the *-logfile* option to specify where the output is to be recorded. If file redirection is absolutely needed, include the *-nolog* option to suppress the generation of the log file that the tool would normally create.

### **-MAXdelays**

Apply the maximum delay value from a timing triplet in the form *min:typ:max* that appears in a *specify* block in the Verilog description.

This option also selects the maximum delay value if the *min:typ:max* value appears in the SDF file while annotating to Verilog or to VITAL unless an SDF-specific construct is used to override it. For example, if you use *-maxdelays* on the command line, but specify *MINIMUM* in an SDF command file (*MTM\_CONTROL = "MINIMUM"*), in an SDF configuration file (*MTM = MINIMUM;*), or in the *\$sdf\_annotation* task, the maximum values in the *specify* block will be used, but the minimum values in the SDF file will be used.

Example:

```
% ncelab -maxdelays top_mod
```

See “[Specifying the Delay Mode for LMG Hardware Models](#)” on page 950 for information on specifying delays for a design that includes an LMC hardware model.

### **-MESSAGES**

Print informative messages during execution. During elaboration, the messages also provide some statistical information about the design hierarchy.

Example:

```
% ncelab -messages top
```

By default, elaborator messages are printed to a log file called `ncelab.log`. Use `-logfile` to rename the log file. Use `-nolog` if you don't want a log file.

Messages are also printed to the screen by default. Use `-nostdout` if you want to suppress printing to the screen.

For Verilog portions of the design, you can use the `-libverbose` option to display additional messages that provide information on the binding of instances to compiled design units.

### **-Mindelays**

Apply the minimum delay value from a timing triplet in the form `min:typ:max` that appears in a specify block in the Verilog description.

This option also selects the minimum delay value if the `min:typ:max` value appears in the SDF file while annotating to Verilog or to VITAL unless an SDF-specific construct is used to override it. For example, if you use `-mindelays` on the command line, but specify `MAXIMUM` in an SDF command file (`MTM_CONTROL = "MAXIMUM"`), in an SDF configuration file (`MTM = MAXIMUM ;`), or in the `$sdf_annotation` task, the minimum values in the specify block will be used, but the maximum values in the SDF file will be used.

Example:

```
% ncelab -mindelays top_mod
```

See “[Specifying the Delay Mode for LMG Hardware Models](#)” on page 950 for information on specifying delays for a design that includes an LMC hardware model.

### **-NCError warning\_code**

Increase the severity level of the specified warning message from warning to error. The `warning_code` argument is the message code (mnemonic) that appears in the warning message following the severity code. You can enter the `warning_code` in uppercase or in lowercase.

Example:

By default, the elaborator issues the following warning message if you have a `$sdf_annotation` system task in your HDL, but the SDF file cannot be found:

```
ncelab: *W,CUSFNF: The SDF file "test.sdf" not found..
```

If you want to upgrade this warning message to an error message so that the elaborator will stop, use the following option on the command line:

```
% ncelab -ncerror CUSFNF worklib.top:module
```

You can include multiple `-ncerror` options on the command line.

Using this option can change the behavior of the tool because functions that return errors instead of warnings may behave differently. Warnings that are changed to errors are counted in the error count limit that you specify with the `_errormax` option.

#### **-NCFatal {*warning\_code* | *error\_code*}**

Increase the severity level of the specified warning message or error message from warning or error to fatal. The `warning_code` or `error_code` argument is the message code (mnemonic) that appears in the message following the severity code.

Example:

```
% ncelab -ncfatal LMNOPQ worklib.top:module
```

You can include multiple `-ncfatal` options on the command line.

#### **-NEG\_tchk**

##### **(Verilog only)**

Allow negative values in `$setuphold` and `$recrem` timing checks in the Verilog description and in `SETUPHOLD` and `RECREM` timing checks in SDF annotation.

Beginning with the LDV 3.3 release, negative timing checks are enabled by default, and you do not have to use this option. Use the `_noneg_tchk` option to turn off negative timing checks.

See “[\\$setuphold](#)” on page 707 and “[\\$recrem](#)” on page 719 for more information on negative timing checks.

**Note:** If you use the `-neg_tchk` option, the values of timing check limits cannot be modified using PLI/VPI routines at simulation time.

#### **-NEVerwarn**

Disable printing of all warning messages.

```
% ncelab -neverwarn top
```

To turn off one or more specific warning messages, use `_nowarn`.

### **-NOAutosdf**

#### **(Verilog only)**

Do not perform automatic SDF annotation.

The elaborator recognizes `$$sdf_annotate` system tasks in the design source files, and if the `$$sdf_annotate` system tasks are scheduled to run at time 0 and meet other requirements, annotation is performed automatically. Use the `-noautosdf` option if you do not want to annotate the design.

See [Chapter 17, “SDF Timing Annotation,”](#) for details on SDF annotation.

### **-NOCopyright**

Suppress the printing of the copyright banner.

Because the copyright banner is displayed before any variables in the `hdl.var` file are processed, this option is ignored if you include it with the `NCELABOPTS` variable in an `hdl.var` file.

### **-NODEadcode**

Turns off the “dead” code optimization.

The NC simulators include an optimization that prevents code that does not contribute in any way to the output of the model from running. Because this “dead” code does not run, any run-time errors, such as constraint errors or null access dereferences, that would be generated by the code are not generated. Other simulation differences (for example, with delta cycle counts and active time points) can also occur.

Use the `-nodeadcode` command-line option to turn off this optimization.

### **-NOIpd**

#### **(VHDL only)**

Ignore the input path delays in a VITAL level 1 cell and read the non-delayed input signals directly.

This option causes the elaborator to ignore the lumped interconnect delays that are specified on input ports in the `WIREDELAY` block. Use this option to speed up the simulation of circuits when the input port delays should be ignored.

For distributed delay models (VITAL primitive procedures that have delays on them), VITAL lets you specify a different delay from each input to the output. You may not need this level of accuracy for all of your simulation runs and you may want to compromise on it to improve speed and/or memory in your simulation.

### **-NOLog**

Do not generate a log file. By default, `ncelab` generates a log file called `ncelab.log`.

The `-nolog` option is overridden by the `-logfile` option.

### **-NONEg\_tchk**

Do not allow negative values in `$setuphold` and `$recrem` timing checks in the Verilog description and in `SETUPHOLD` and `RECREM` timing checks in SDF annotation. If you use this option, any negative values in the description or in the SDF annotation are set to 0 and a warning is issued.

See “[\\$setuphold](#)” on page 707 and “[\\$recrem](#)” on page 719 for more information on negative timing checks.

### **-NONNotifier**

#### **(Verilog only)**

Ignore notifiers in timing checks.

See “[Using Notifiers](#)” on page 723 for information on using notifiers.

### **-NOSource**

Ignore source file timestamps when using the `-update` option.

### **-NOSTdout**

Suppress the printing of output to the screen.

The `-nostdout` option does not change what is written to the log file.

### **-NOTImingchecks**

Do not execute timing checks.

This option turns off both Verilog and accelerated VITAL timing checks.

### **-NOVitalaccI**

**(VHDL only)**

SUPPRESS the acceleration of VITAL level 1 compliant cells. Using this option may help you to debug a problem if you are seeing unexpected simulation results.

### **-NOWarn *warning\_code***

Disable the printing of the warning with the specified code. For example, when elaborating, you may know about unconnected signals in your model. While the individual design units or source files may compile without error, the elaborator will generate port mismatch warning messages. If you aren't interested in seeing these messages, use `-nowarn` to turn them off.

The *warning\_code* argument is the message code (mnemonic) that appears in the warning message that follows the error severity code.

Example:

```
% ncelab -nowarn CUVWSP top
```

You can include multiple `-nowarn` options on the command line.

**-NO\_Sdfa\_header**

**(Verilog only)**

Do not print elaborator informational messages that display information contained in the SDF command file. These messages tell you what arguments were used for the various keywords in the command file.

See “[Writing an SDF Command File](#)” on page 791 for information on the SDF command file.

**-NO\_TCHK\_Msg**

Do not display timing check warning messages.

This option turns off messages for both Verilog and accelerated VITAL timing checks.

**-NO\_TCHK\_Xgen**

**(VHDL only)**

Turn off X generation in accelerated VITAL timing checks.

This option has no effect if you use the `-novitalaccl` option.

**-NO\_VPD\_Msg**

**(VHDL only)**

Turn off glitch messages from accelerated VITAL pathdelay procedures.

This option has no effect if you use the `-novitalaccl` option.

**-NO\_VPD\_Xgen**

**(VHDL only)**

Turn off X generation in accelerated VITAL pathdelay procedures.

This option has no effect if you use the `-novitalaccl` option.

### **-NTc\_warn**

Print convergence warnings for negative timing checks for both Verilog and VITAL if delays cannot be calculated given the current limit values. By default, warnings are not printed.

See “[Negative Timing Check Limits in \\$setuphold and \\$recrem](#)” on page 728 for details on how delays are calculated when negative limits are used.

### **-OMicheckinglevel *checking\_level***

Specify OMI checking level. The *checking\_level* argument can be:

- max—Maximum checking level. Use this level for early integration testing and to debug problems.
- std—Standard checking level. This is the default.
- min—Minimum checking level. Select this level to achieve higher performance after problems have been debugged.

See “[The Open Model Interface \(OMI\)](#)” on page 952 for details on OMI.

### **-Overwrite**

#### **(VHDL only)**

Overwrite an existing configuration file that has the same name.

By default, the elaborator does not overwrite an existing configuration file that already exists in a library if the file has the same name as the configuration file that you are generating. Use the `-overwrite` option if you want to generate a configuration file that has the same name as an existing configuration file.

You can use the `-overwrite` option only when you are generating a VHDL configuration file with the `-conffile` option.

**-PAthpulse**

**(Verilog only)**

Enable PATHPULSE\$ specparams, which are used to set module path pulse control on a specific module or on specific paths within modules.

See “[Setting Pulse Controls](#)” on page 280 for more information.

**-PLINOOptwarn**

**(Verilog only)**

Display only one warning message the first time that a PLI read, write, or connectivity access violation is detected.

By default, the elaborator displays all of the warning and error messages that are generated when an error is detected due to a PLI access violation. Use this option to suppress the display of these access violation messages. If you use this option, a warning message is displayed once, when the first read or write access violation is detected. The message is displayed again if an access violation is detected after a reset or a restart has been executed.

Example:

```
% ncelab -plinooptwarn top
```

**-PLINOWarn**

**(Verilog only)**

Suppress the display of PLI warning and error messages. These messages are displayed by default.

Example:

```
% ncelab -plinowarn alu_16
```

## **-PReserve**

### **(VHDL only)**

Preserve resolution functions on signals with only one driver.

This option allows reflexive signal calls to the resolution function; otherwise, these calls are removed for simulation performance improvement.

A resolved signal is called *reflexive* when it has only one source and the value of the signal is defined to be the same as that source. This case is common. Type conversions are not required to resolve reflexive signals because the output is the same as the input.

The elaborator identifies reflexive signals and removes the call to the resolution function in the simulator. Removing this function improves the performance of the signal evaluation process.

To always call resolution functions, use `-preserve`.

## **-PROmpt**

### **(VHDL only)**

Display a list of all architectures that are available for an entity and ask the user to select one.

You can use the `-prompt` option only when you are generating a configuration file with the `-conffile` option.

If an explicit binding is not specified in the VHDL source code, the elaborator, by default, selects the most-recently analyzed architecture for an entity. You can override this selection by including the `-prompt` option on the command line. If you use this option, the elaborator displays a list of all available architectures for an entity. You can then select the architecture that you want for an entity.

See “[Using ncelab to Generate a VHDL Configuration File](#)” on page 228 for an example.

You can also override the selection of the most-recently analyzed architecture by using the `-usearch` option to specify a priority list of architectures.

**-PULSE\_E error\_percent**

**(Verilog only)**

Set the percentage of delay for the pulse error limit for both module paths and interconnect. If the -pulse\_int\_e option is also used, this option applies only to module paths.

See “[Setting Pulse Controls](#)” on page 280 for more information.

**-PULSE\_INT\_E error\_percent**

**(Verilog only)**

Set the percentage of delay for the pulse error limit for interconnect only.

See “[Setting Pulse Controls](#)” on page 280 for more information.

**-PULSE\_INT\_R reject\_percent**

**(Verilog only)**

Set the percentage of delay for the pulse reject limit for interconnect only.

See “[Setting Pulse Controls](#)” on page 280 for more information.

**-PULSE\_R reject\_percent**

**(Verilog only)**

Set the percentage of delay for the pulse reject limit for both module paths and interconnect. If the -pulse\_int\_r option is also used, this option applies only to module paths.

See “[Setting Pulse Controls](#)” on page 280 for more information.

**-Relax**

**(VHDL only)**

Allow design units to be visible for default binding when those design units exist in a library that has not been made visible with a LIBRARY declaration in the VHDL source and when the design units do not exist in the library that has been defined as the work library.

By default, NC-VHDL adheres to a strict interpretation of the VHDL LRM, which states that you must use `LIBRARY` statements in the source code to provide visibility to the declarative region that an unbound instance resides in. To bind component instances to compiled design units in the libraries, the elaborator:

1. Uses explicit binding indications.
2. If there is no explicit binding indication, the elaborator tries to bind the component to (in order):
  - ❑ A design unit in a library made visible in a `USE` clause
  - ❑ A design unit in a library that has been explicitly declared
  - ❑ A design unit in the work library

If a binding cannot be found, the elaborator generates an error.

Use the `-relax` option to extend default binding to all libraries that are defined in the `cds.lib` file. If a binding has not been found, the elaborator opens the `cds.lib` file and searches all of the libraries that are defined in the file and that have not already been searched. The search stops when the elaborator finds a component that has the same name or after all libraries have been searched and binding has failed.

When using the `cds.lib` file for visibility, the elaborator searches the libraries in the order in which they appear, and `cds.lib` files are searched sequentially in the order that they appear in the main `cds.lib` file. For example, given the following `cds.lib` file, the search order would be: `foo3, foo1, foo2, foo4`.

```
# File: cds.lib
INCLUDE my_cds.lib
DEFINE foo1 ./foo
DEFINE foo2 ./foo2
INCLUDE my_other_cds.lib

# File: my_cds.lib
DEFINE foo3 ./foo3

# File: my_other_cds.lib
DEFINE foo4 ./foo4
```

If a library is redefined, the new definition supersedes the old definition. For example, given the following `cds.lib` file, the order of libraries to be searched is `lib2, lib1, lib3`.

```
# File: cds.lib
DEFINE lib1 lib1
DEFINE lib2 lib2
DEFINE lib1 lib1
DEFINE lib3 lib3
```

For a mixed-language design in which the top-level is VHDL, the elaborator will select a VHDL unit over a Verilog unit that has the same name, even if the VHDL unit is in a library that is listed after the library that contains the Verilog unit. If the top-level is Verilog, the elaborator will select a Verilog unit over a VHDL unit that has the same name.

#### **-SDF\_Cmd\_file *sdf\_command\_file***

Use the specified SDF command file to control SDF annotation.

For VITAL SDF annotation, you must write an SDF command file and then include the command file with the `-sdf_cmd_file` option. For Verilog, you can annotate by using `$sdf_annotate` or by using an SDF command file.

See [Chapter 17, “SDF Timing Annotation,”](#) for details on SDF annotation.

#### **-SDF\_NOCHECK\_Celltype**

**(Verilog only)**

Disable celltype validation between the SDF annotator and the Verilog description. By default, the annotator checks the type that is specified in the `CELLTYPE` construct against the module name in the description. If there is a mismatch, a warning is generated and no annotation to that module instance is performed.

See [Chapter 17, “SDF Timing Annotation,”](#) for details on SDF annotation.

#### **-SDF\_NO\_Warnings**

Do not report warning messages from the SDF annotator.

See [Chapter 17, “SDF Timing Annotation,”](#) for details on SDF annotation.

### **-SDF\_Precision precision**

Round the precision of timing values in the compiled SDF file.

The SDF compiler (*ncsdfc*) compiles the SDF file with a precision of 1 fs. Use the `-sdf_precision` option if you want to specify a coarser precision. Specifying a coarser precision can improve simulation performance.

The *precision* argument consists of an integer and a time unit. The integer can be 1, 10, or 100. The time unit can be `fs`, `ps`, `ns`, `us`, or `s`. No space is allowed between the integer and the time unit.

In the following command, the `-sdf_precision` option specifies a precision of 100 picoseconds for timing values.

```
% ncelab -sdf_precision 100ps
```

Timing values in the compiled SDF file are rounded to the nearest 100 ps. For example, the timing value in the following `IOPATH` statement is rounded to 6.1.

```
(IOPATH in out (6.127))
```

The timing values in the following `IOPATH` statement are rounded to 6.1 : 9.6 : 15.0.

```
(IOPATH in out (6.127:9.554:15.031))
```

### **-SDF\_Verbose**

Include detailed information in the SDF log file.

You specify the SDF log file with the *log\_file* argument of the `$sdf_annotation` system task or with the `LOG_FILE` statement in an SDF command file.

See “[\\$sdf\\_annotation System Task](#)” on page 803 for information on the arguments of the `$sdf_annotation` task. See “[Writing an SDF Command File](#)” on page 791 for details on the SDF command file.

### **-SDF\_Worstcase\_rounding**

#### **(Verilog only)**

For timing values in the SDF file, truncate the min value, round the typ value, and round up the max value. For example, using this option changes the annotated timing values in the following `IOPATH` statement to 0 : .1 : .1 (assuming a precision of .1).

```
(IOPATH in out (.05:.05:.03))
```

How a single timing value is treated depends on the command-line option that you use. For example, the timing value in the following IOPATH statement is annotated as 0 for -mindelays, and as .1 for -typdelays and -maxdelays.

```
(IOPATH in out (.05))
```

### **-SNAPSHOT snapshot\_name**

Use the specified name for the simulation snapshot. Use this option to give different elaborations of your design unique snapshot names.

The *snapshot\_name* argument is a Library.Cell:View specification. The default, if the full specification is not given, is the cell name.

```
[Library.]Cell[:View]
```

If you do not specify the *-snapshot* option, the snapshot name is the name of the top-level design unit that you specified on the command line. If you specify more than one Verilog top-level module on the command line, the snapshot name is the name of the first top-level module.

Example:

```
% ncelab alu_16 -snapshot alu16_vcd
```

**Note:** In the Leapfrog VHDL simulator, the elaborator (ev) generates snapshots that have a fourth-level name in the library structure (for example, L.E:A/SIM). You cannot specify a snapshot name that includes a slash character ( / ) with the ncelab *-snapshot* option. For example, the following command generates an error message:

```
% ncelab -messages -snapshot worklib.alu_16:behave/SIM alu_16:behave
```

### **-STATUS**

Print statistics on memory and CPU usage after elaboration.

The following example shows the output of the *-status* option:

```
ncelab: Memory Usage - 8.4M program + 3.9M data = 12.3M total
ncelab: CPU Usage - 1.6s system + 1.0s user = 2.6s total (2.9s, 92.2% cpu)
```

**-TFile *timing\_file***

**(Verilog only)**

Use the specified timing file. A timing file is a text file that lets you turn off timing for particular instances or portions of a design. See “[Disabling Timing in Selected Portions of a Design](#)” on page 267 for details on writing and using a timing file.

**-Timescale ‘*time\_unit / time\_precision*’**

**(Verilog only)**

Set the default timescale for Verilog modules that do not have a timescale set.

With the NC-Verilog simulator, as with the Verilog-XL simulator, you must include a timescale for all module definitions if any module has been compiled with a ‘timescale compiler directive. Verilog-XL generates an error for modules that do not have a ‘timescale directive. With the NC-Verilog simulator, the elaborator generates the error message.

You can avoid this error by:

- Adding a ‘timescale directive to each unit (perhaps after a ‘resetall directive).
- Including a ‘timescale directive in the first unit that you list on the compile command, and then making sure that other units do not override its effect. For example, a subsequent unit that has a ‘resetall directive must also have a ‘timescale directive.
- Using the –timescale option to specify a default timescale for modules that do not otherwise have one.

Using the –timescale option is useful in situations where you do not want to, or cannot, edit files that are generated by other tools or that are provided by library vendors. For example, a synthesis tool may generate a structural netlist that models the device as Verilog gates connected by wires. No timing information is needed at this level, and each subcomponent may have a ‘timescale directive. You can use the –timescale option to specify a timescale for the top-level module.

The format of the argument to the –timescale option is the same as that for the ‘timescale directive. Enclose the argument in single quotation marks.

**Example:**

```
% ncelab -timescale '1 ns / 1 ps'
```

### **-TYpdelayS**

Apply the typical delay value from a timing triplet in the form min:typ:max that appears in a specify block in the Verilog description.

This option also selects the typical delay value if the min:typ:max value appears in the SDF file while annotating to Verilog or to VITAL unless an SDF-specific construct is used to override it. For example, if you use `-typdelays` on the command line, but specify MAXIMUM in an SDF command file (`MTM_CONTROL = "MAXIMUM"`), in an SDF configuration file (`MTM = MAXIMUM ;`), or in the `$sdf_annotation` task, the typical values in the specify block will be used, but the maximum values in the SDF file will be used.

Example:

```
% ncelab -typdelays top_mod
```

See “[Specifying the Delay Mode for LMG Hardware Models](#)” on page 950 for information on specifying delays for a design that includes an LMC hardware model.

### **-UPdate**

Automatically recompile any out-of-date design units and then re-elaborate the design.

### **-USe5x4vhdl**

**(VHDL only)**

Use 5.X configurations for elaborating VHDL hierarchies.

A 5.X configuration is an ASCII text file, usually written with a tool such as the Hierarchy Editor, that specifies the rules that the elaborator is to use for selecting design units out of design libraries for binding to instances in a design hierarchy. 5.x configurations are used by Cadence® AMS simulator users.

The `-use5x4vhdl` option affects the set of rules that is used to resolve a VHDL instance during elaboration.

- If you do not include the `-use5x4vhdl` option, the elaborator first uses VHDL language rules (configuration declarations, configuration specifications, entity aspect specifications) and then the default VHDL binding rules to determine a binding.
- If you include the option, the elaborator first uses VHDL language rules to determine a binding. It then uses the binding rules specified in a 5.x configuration specification before using the default binding rules.

### **-USEArch *priority\_list\_of\_architectures***

**(VHDL only)**

Use the specified priority list of architectures when selecting an architecture for an entity.

You can use the `-usearch` option only when you are generating a configuration file with the `-conffile` option.

If an explicit binding is not specified in the VHDL source code, the elaborator, by default, selects the most-recently analyzed architecture for an entity. You can override this selection by including the `-usearch` option on the command line.

The *priority\_list\_of\_architectures* argument is a list of architectures. Separate the items on the list with a comma. No spaces are allowed in the list.

Example:

```
ncelab -conffile abc.vhd WORKLIB.TEST_DFF:TEST_BENCH_DFF  
      -usearch DFF_ARCH_1,DFFARCH_2
```

The `-usearch` option is useful for quickly generating a configuration file with different bindings. For example, suppose that you specify the following option on the command line:

```
-usearch rtl,behav
```

In this case, the elaborator selects the architecture called `rtl` for all entities that have an architecture called `rtl`. For all other entities, the elaborator selects the architecture called `behav`. If the priority list is exhausted, the most-recently analyzed architecture is selected.

### **-V93**

**(VHDL only)**

Enable VHDL-93 features. See “[Features Included from the IEEE 1076-1993 Standard](#)” in the *NC-VHDL Simulator Help* for a list of supported VHDL-93 features.

### **-VIPDMAx**

**(VHDL only)**

During VITAL SDF annotation, select the maximum delay value if more than one interconnect specification maps to the same interconnect path delay generic.

By default, the SDF annotator maps every interconnect construct that has the same destination to one `tipd` generic that is associated with the destination port. When more than one construct maps to a given generic, the annotator sets the value of the generic to the last interconnect delay that it encounters. Use the `-vipdmax` option to select the maximum delay value.

Use the `-intermod_path` option if you want to specify unique delays for each source-load path during VITAL SDF annotation. See “[VITAL SDF Annotation](#)” on page 790 for details on VITAL SDF annotation.

You cannot use the `-vipdmax` option with the `-intermod_path` option.

#### **-VIPDMin**

##### **(VHDL only)**

During VITAL SDF annotation, select the minimum delay value if more than one interconnect specification maps to the same interconnect path delay generic.

By default, the SDF annotator maps every interconnect construct that has the same destination to one `tipd` generic that is associated with the destination port. When more than one construct maps to a given generic, the annotator sets the value of the generic to the last interconnect delay that it encounters. Use the `-vipdmin` option to select the minimum delay value.

Use the `-intermod_path` option if you want to specify unique delays for each source-load path during VITAL SDF annotation. See “[VITAL SDF Annotation](#)” on page 790 for details on VITAL SDF annotation.

You cannot use the `-vipdmin` option with the `-intermod_path` option.

#### **-VErsion**

Print the version of the elaborator and exit.

This option is ignored if you include it with the `NCELABOPTS` variable in an `hdl.var` file.

**-Work *work\_library***

**(VHDL only)**

Use the specified library as the work library in default binding. This option overrides the setting for the WORK variable in the `hdl.var` file.

## Example ncelab Command Lines

The following command includes the `-messages` option, which prints elaborator messages.

```
% ncelab -messages top
```

The following example includes the `-logfile` option, which renames the log file from `ncelab.log` to `top_elab.log`.

```
% ncelab -messages -logfile top_elab.log top
```

In the following example, `-errormax 10` tells the elaborator to abort after 10 errors.

```
% ncelab -messages -errormax 10 top
```

The following example uses the `-file` option to include a file called `ncelab.args`, which contains a set of elaborator command-line options.

```
% ncelab -file ncelab.args top
```

The following example uses the `-snapshot` option to name the snapshot `topsnap`. When the simulator is invoked, this name should be used with the `ncsim` command.

```
% ncelab top -snapshot topsnap
```

In the following example, `-nowarn` is used to suppress the printing of a specific error message. The argument to the option is the mnemonic for the message.

```
% ncelab top
ncelab: v1.1.(p2): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
b_2bit_adder under_test (sum, c_out, bus_a, bus_b, c_in);
          |
ncelab: *E,CUVWLP (2bit_adder_test.v,7|22): Too many module port connections.
% ncelab -nowarn CUVWLP top
ncelab: v1.1.(p2): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
```

The following example uses the `-sdf_cmd_file` option to specify an SDF command file called `dcache_sdf.cmd`. Using this option overrides the automatic SDF annotation to Verilog portions of the design. The command file contains commands that control SDF annotation. See [Chapter 17, “SDF Timing Annotation,”](#) for details on SDF annotation.

```
% ncelab -messages -sdf_cmd_file dcache_sdf.cmd top
```

## Using ncelab to Generate a VHDL Configuration File

The following examples show you how to generate a VHDL configuration file by running *ncelab*.

The following source file is used in the examples:

```
-- File: test.vhd
-- Testbench entity
entity TEST_DFF is
end TEST_DFF;

-- Testbench architecture body
architecture TEST_BENCH_DFF of TEST_DFF is
    component DFF is
        port (D, CLK : in BIT; Q, QBAR : out BIT);
    end component DFF;
    signal DIN, CP, Q, QBAR : BIT;
begin

    DFF_DUT : DFF port map (D => DIN, CLK => CP, Q => Q, QBAR => QBAR);

    CLOCK : process (CP)
    begin
        CP <= not CP after 10 ns;
    end process CLOCK;

end TEST_BENCH_DFF;

entity DFF is
    port ( D, CLK: in BIT; Q, QBAR : out BIT);
end DFF;

architecture DFF_ARCH_1 of DFF is
begin

    process (D, CLK)
    begin
        if D'EVENT then
            assert false
                report "Event Occurred on D !!!"
                severity NOTE;
        end if;
    end process;
end;
```

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

```
    end if;
    end process;

end DFF_ARCH_1;
architecture DFF_ARCH_2 of DFF is
begin

process (D, CLK)
begin
if (CLK = '1') and (CLK'EVENT) then
    Q <= D;
    QBAR <= not D;
end if;
end process;

end DFF_ARCH_2;
```

In the following sequence of commands, the source file `test.vhd` is compiled, and then `ncelab` is invoked with the `-conffile` option to generate a VHDL configuration file for the design unit `WORKLIB.TEST_DFF:TEST_BENCH_DFF`. The `-compile` option is included on the `ncelab` command line to automatically compile the configuration file.

By default, `ncelab` generates a configuration file in hierarchical format. The name of the configuration is `cfg_entity_architecture` (`cfg_DFF_DFF_ARCH_2` in this example). Because there is no explicit binding specified in the source file, the most-recently analyzed architecture (`DFF_ARCH_2`) is selected.

```
% ncvhdl -nocopyright -v93 test.vhd
% ncelab -messages -conffile abc.vhd WORKLIB.TEST_DFF:TEST_BENCH_DFF -compile
ncelab: v3.20.(p1): (c) Copyright 1995 - 2000 Cadence Design Systems, Inc.
      Elaborating the design hierarchy:
      Wrote output file WORKLIB_abc.vhd
      Compiling configuration file(s)
ncvhdl -messages -work WORKLIB WORKLIB_abc.vhd
ncvhdl: v3.20.(p1): (c) Copyright 1995 - 2000 Cadence Design Systems, Inc.
WORKLIB_abc.vhd:
      errors: 0, warnings: 0
WORKLIB.CFG_DFF_DFF_ARCH_2 (configuration):
      streams: 1, words: 3
WORKLIB.CFG_TEST_DFF_TEST_BENCH_DFF (configuration):
      streams: 2, words: 48
```

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

```
% more worklib_abc.vhd
--
-- Configuration for library WORKLIB
-- Configuration Model: HIERARCHICAL
--
library WORKLIB;
configuration cfg_DFF_DFF_ARCH_2 of DFF is
    for DFF_ARCH_2
        end for;
end cfg_DFF_DFF_ARCH_2;

library WORKLIB;
configuration cfg_TEST_DFF_TEST_BENCH_DFF of TEST_DFF is
    for TEST_BENCH_DFF
        for DFF_DUT: DFF use configuration WORKLIB.cfg_DFF_DFF_ARCH_2;
        end for;
    end for;
end cfg_TEST_DFF_TEST_BENCH_DFF;
```

In the following example, the **-confflat** option is used to generate a flat configuration file.

```
% ncelab -conffile xyz.vhd WORKLIB.TEST_DFF:TEST_BENCH_DFF -confflat
ncelab: v3.20.(p1): (c) Copyright 1995 - 2000 Cadence Design Systems, Inc.
% more xyz.vhd
--
-- Configuration for top level unit WORKLIB.TEST_DFF:TEST_BENCH_DFF
-- Configuration Model: FLAT
--
library WORKLIB;
configuration cfg_TEST_DFF_TEST_BENCH_DFF of TEST_DFF is
    for TEST_BENCH_DFF
        for DFF_DUT: DFF use entity WORKLIB.DFF(DFF_ARCH_2);
            for DFF_ARCH_2
                end for;
            end for;
        end for;
    end for;
end cfg_TEST_DFF_TEST_BENCH_DFF;
```

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

The following command includes the `-prompt` option. This option displays the architectures that are available for entity DFF and prompts you to select one. In this example, the architecture labeled 0 is selected. The `-overwrite` option is also included to overwrite the configuration file called abc.vhd that was generated in the first example above.

```
% ncelab -mess -conffile abc.vhd -overwrite WORKLIB.TEST_DFF:TEST_BENCH_DFF  
-prompt
```

```
ncelab: v3.20.(p1): (c) Copyright 1995 - 2000 Cadence Design Systems, Inc.
```

```
    Elaborating the design hierarchy:
```

```
ncelab> Enter the number of the architecture to be bound to entity WORKLIB.DFF
```

```
0: DFF_ARCH_1
```

```
1: DFF_ARCH_2
```

```
0
```

```
The selected architecture is DFF_ARCH_1
```

```
Wrote output file WORKLIB_abc.vhd
```

```
%
```

## hdl.var Variables

The following variables are used by *ncelab*.

See “[The hdl.var File](#)” on page 118 for more information on the `hdl.var` file.

■ **NCELABOPTS**

Sets elaborator command-line options. A top-level module name(s) can also be included.

```
DEFINE NCELABOPTS -messages -errormax 10
```

■ **LIB\_MAP (Verilog)**

This variable is used by both the *ncvlog* compiler and by the elaborator. The compiler uses the definition of the variable to map files and directories to library names. Use the plus sign ( + ) to specify other files or directories that are not explicitly specified. The elaborator uses this variable to establish the list of libraries to search, and the order in which to search them, when resolving instances.

Example:

```
DEFINE LIB_MAP ( myfile.v => mylib, \
                 yourfile.v => yourlib, \
                 ./source    => source, \
                 +           => worklib )
```

■ **VIEW\_MAP (Verilog)**

This variable is used by both the *ncvlog* compiler and by the elaborator. The compiler uses the definition of the variable to map file extensions to view names. The elaborator uses this variable to establish the list of views to search, and the order in which to search them, when resolving instances.

Example:

```
DEFINE VIEW_MAP ( .v => behav, .rtl => rtl, .gate => gate )
```

■ **WORK**

Defines the work library. This variable is used by the elaborator only for VHDL default binding. Using the `-work` option overrides the setting of this variable.

## How Modules and UDPs Are Resolved During Elaboration

One of the most important operations that occurs during elaboration is *binding* (or *linking*). Binding is the process of selecting which design units are instantiated at each node of the hierarchy. Each module or UDP that is instantiated in another, higher-level, module is bound to a particular Lib.Cell:View.

**Note:** The term *binding* refers to the process of selecting a particular Lib.Cell:View for each module or UDP that is instantiated in another, higher-level, module. The binding rules that are described in this section do not apply to the top-level module(s) that you specify on the command line. See “[Overview](#)” on page 181 for information on the rules for selecting a Lib.Cell:View for top-level modules.

This section discusses:

- The default binding mechanism. See “[The Default Binding Mechanism](#)” on page 234.
- The `-binding` option, which is used to force the binding of a cell to a particular library and view. The `-binding` option overrides the default binding mechanism. See “[The -binding Option](#)” on page 239.
- The ``uselib` compiler directive, which overrides the default binding mechanism and the `-binding` option. See “[The ‘uselib’ Compiler Directive](#)” on page 242.

## The Default Binding Mechanism

The binding rules are as follows:

1. For a particular module, if an instance has already been bound during the current elaboration, use the same binding. Otherwise, proceed to step 2.
2. Using the libraries that are listed with the `LIB_MAP` variable, and beginning with the library where the parent module was found, search the view names that are listed with the `VIEW_MAP` variable in order, beginning with the view that has the same view name as the parent module. If a view that has the same name as the view of the parent module is found, use that binding.
3. If no binding is found, continue with the next view in the `VIEW_MAP` variable. Continue searching the views in order, wrapping around to the first view, if necessary.
4. If no binding is found, move on to the next library listed in the `LIB_MAP` variable and repeat the view search using the same steps.
5. If no binding is found, search the library where the parent module was found to determine a possible binding.
  - If one binding exists, use it.
  - If more than one binding exists, exit with an error.
6. If no binding exists, search all known libraries.
  - If one binding exists, use it.
  - If more than one binding exists, exit with an error.

The following example shows the error message that is generated when multiple bindings exist. Note that all possible bindings are listed as part of the error message.

```
ncelab: *E,MULTBI: Possible bindings for instance of module/UDP 'bot' in  
'worklib.top:module' are:  
      worklib.bot:v1  
      worklib.bot:v2
```

- If no binding exists, exit with an error.

If no binding is possible, *ncelab* generates an error message similar to the following example:

```
ncelab: *E,CUVMUR: instance of module/UDP 'bot' is unresolved in  
'worklib.top:module'.
```

If the `LIB_MAP` variable has not been defined, *ncelab* searches the libraries listed in the `cds.lib` file. The libraries are searched in order, beginning with the library where the parent module was found, and wrapping around to the first library, if necessary.

If you have not defined the `VIEW_MAP` variable, *ncelab* searches only the default views (`module` and `udp`). If you define a `VIEW_MAP` variable, and you want the elaborator to search for the default views, you must have an entry in the `VIEW_MAP` variable for `module` and `udp`.

Use the `-libverbose` option to display binding messages. For example,

```
% ncelab -libverbose top
```

The following three examples illustrate the binding rules. The following files are used in the examples:

```
# hdl.var file
DEFINE VIEW_MAP ( .v      => behav, \
                  .rtl    => rtl, \
                  .gate   => gate )
DEFINE LIB_MAP ( ./designlib => designlib, \
                  ./source   => source, \
                  +          => worklib )

// File top.rtl                      // File ./designlib/foo.v
module top ();                         module foo ();
  foo a ();                            ...
  foo b ();                            ...
endmodule                           endmodule
```

### Example 1:

In the first example, `module top` in `top.rtl` is compiled into `worklib.top:rtl`, and `module foo` in `./designlib/foo.v` is compiled into `designlib.foo:behav` using the definitions of the `LIB_MAP` and `VIEW_MAP` variables.

*ncelab* first searches the library where the parent module (`top`) was found (`worklib`) for a view that has the same view name as the parent module (`rtl`). It then searches `worklib` for a `gate` or `behav` view. No binding is found, so *ncelab* moves to the next library that is listed with the `LIB_MAP` variable (`designlib`), where it looks first for a view called `rtl` and then for a view called `gate` before finding view `behav`.

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

```
;# Compile top.rtl. The VIEW_MAP variable maps files with a .rtl extension to
;# a view called rtl. Compilation produces worklib.top:rtl.
% ncvlog -messages top.rtl
ncvlog: v2.2.(p1): (c) Copyright 1995-1999 Cadence Design Systems, Inc.
file: top.rtl
    module worklib.top:rtl
        errors: 0, warnings: 0

;# Compile ./designlib/foo.v. The LIB_MAP variable maps files in this directory
;# to the designlib library. The VIEW_MAP variable maps files with a .v
;# extension to a view called behav. Compilation produces designlib.foo:behav.
% ncvlog -messages ./designlib/foo.v
ncvlog: v2.2.(p1): (c) Copyright 1995-1999 Cadence Design Systems, Inc.
file: ./designlib/foo.v
    module designlib.foo:behav
        errors: 0, warnings: 0

;# Elaborate the top-level module, top. Use the -libverbose option to display
;# messages. ncelab looks in worklib for the same view as the parent (rtl),
;# then for a gate or behav view. It then does the same view search in designlib.
% ncelab -messages -libverbose top
ncelab: v2.2.(p1): (c) Copyright 1995-1999 Cadence Design Systems, Inc.
Elaborating the design hierarchy:
Resolving module/udp 'foo' at 'top.a'.
    Caching library 'worklib' .... Done
    library: 'worklib' views: 'rtl' 'gate' 'behav' -> not found
    Caching library 'designlib' .... Done
    library: 'designlib' views: 'rtl' 'gate' 'behav' -> found
Resolved module/udp 'foo' at 'top.a' to 'designlib.foo:behav'.
Resolved module/udp 'foo' at 'top.b' to 'designlib.foo:behav'.
...
Writing initial simulation snapshot: worklib.top:rtl
```

#### **Example 2:**

In the next example, module top in top.rtl is compiled into worklib.top:rtl. Module foo in ./designlib/foo.v is compiled into designlib.foo:foo using the -view command-line option.

*ncelab* first searches the library where the parent module (top) was found (worklib) for a view that has the same view name as the parent module (rtl). It then searches worklib for a gate or behav view. No binding is found, so *ncelab* moves to the next library that is listed with the LIB\_MAP variable (designlib) and then to the third library (source).

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

*ncelab* then searches for a possible binding in the library where the parent module (`top`) was found (`worklib`). When it does not find a possible binding, *ncelab* searches all known libraries. One view (`designlib.foo:foo`) is found in the library `designlib`.

```
;# Compile top.rtl. The VIEW_MAP variable maps files with a .rtl extension to
;# a view called rtl. Compilation produces worklib.top:rtl.
% ncvlog -messages top.rtl
ncvlog: v2.2.(p1): (c) Copyright 1995-1999 Cadence Design Systems, Inc.
file: top.rtl
    module worklib.top:rtl
        errors: 0, warnings: 0

;# Compile ./designlib/foo.v. The LIB_MAP variable maps files in this directory
;# to the designlib library. The -view option creates a view called foo.
;# Compilation produces designlib.foo:foo.
% ncvlog -messages ./designlib/foo.v -view foo
ncvlog: v2.2.(p1): (c) Copyright 1995-1999 Cadence Design Systems, Inc.
file: ./designlib/foo.v
    module designlib.foo:foo
        errors: 0, warnings: 0

;# Elaborate the top-level module, top. ncelab looks in worklib for the same
;# view as the parent (rtl), then for a gate or behav view. It then does the
;# same view search in designlib, and then in source. ncelab then searches
;# worklib for a possible binding. When ncelab does not find a possible binding,
;# it searches all known libraries. One view (designlib.foo:foo) is found in
;# library designlib.
% ncelab -messages -libverbose top
ncelab: v2.2.(p1): (c) Copyright 1995-1999 Cadence Design Systems, Inc.
Elaborating the design hierarchy:
Resolving module/udp 'foo' at 'top.a'.
    Caching library 'worklib' .... Done
    library: 'worklib' views: 'rtl' 'gate' 'behav' -> not found
        Caching library 'designlib' .... Done
    library: 'designlib' views: 'rtl' 'gate' 'behav' -> not found
        Caching library 'source' .... Done
    library: 'source' views: 'rtl' 'gate' 'behav' -> not found
Resolved module/udp 'foo' at 'top.a' to 'designlib.foo:foo'.
Resolved module/udp 'foo' at 'top.b' to 'designlib.foo:foo'.
...
...
Writing initial simulation snapshot: worklib.top:rtl
```

### **Example 3:**

In the third example, module top in top.rtl is compiled into worklib.top:rtl. Module foo in ./designlib/foo.v is compiled twice using the -view command-line option: into designlib.foo:foo and into designlib.foo:bar.

Using the search mechanism described above, *ncelab* finds two possible bindings: designlib.foo:foo and designlib.foo:bar. *ncelab* generates error messages saying that it found multiple bindings and that no binding was possible.

```
;# Compile top.rtl. The VIEW_MAP variable maps files with a .rtl extension to
;# a view called rtl. Compilation produces worklib.top:rtl.
% ncvlog -messages top.rtl
ncvlog: v2.2.(p1): (c) Copyright 1995-1999 Cadence Design Systems, Inc.
file: top.rtl
    module worklib.top:rtl
        errors: 0, warnings: 0

;# Compile ./designlib/foo.v. The LIB_MAP variable maps files in this directory
;# to the designlib library. The -view option creates a view called foo.
;# Compilation produces designlib.foo:foo.
% ncvlog -messages ./designlib/foo.v -view foo
ncvlog: v2.2.(p1): (c) Copyright 1995-1999 Cadence Design Systems, Inc.
file: ./designlib/foo.v
    module designlib.foo:foo
        errors: 0, warnings: 0

;# Compile ./designlib/foo.v. The LIB_MAP variable maps files in this directory
;# to the designlib library. The -view option creates a view called bar.
;# Compilation produces designlib.foo:bar.
% ncvlog -messages ./designlib/foo.v -view bar
ncvlog: v2.2.(p1): (c) Copyright 1995-1999 Cadence Design Systems, Inc.
file: ./designlib/foo.v
    module designlib.foo:bar
        errors: 0, warnings: 0

% ncelab -messages -libverbose top
ncelab: v2.2.(p1): (c) Copyright 1995 - 1999 Cadence Design Systems, Inc.
    Elaborating the design hierarchy:
Resolving module/udp 'foo' at 'top.a'.
    Caching library 'worklib' .... Done
    library: 'worklib' views: 'rtl' 'gate' 'behav' -> not found
    Caching library 'designlib' .... Done
    library: 'designlib' views: 'rtl' 'gate' 'behav' -> not found
```

```
Caching library 'source' ..... Done
library: 'source' views: 'rtl' 'gate' 'behav' -> not found
ncelab: *E,MULTBI: Possible bindings for instance of module/UDP 'foo' in
'worklib.top:rtl' are:
    designlib.foo:foo
    designlib.foo:bar
ncelab: *E,CUVMUR: instance of module/UDP 'foo' is unresolved in 'worklib.top:rtl'.
```

## The **-binding** Option

Use the **-binding** option to force the binding of a cell to a particular library and view. The syntax is:

```
-binding [lib.]cell[:view]
```

Using this option overrides the default view and library search mechanism. The specified cell is bound to the library and view that you specify.

Remember that once the first instance has been resolved, all instances of the same module or UDP are resolved the same way. Use the ``uselib` compiler directive to force different bindings for modules or UDPs with the same name. See “[The ‘uselib Compiler Directive’](#)” on page 242.

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

The following example shows you how to use the `-binding` option to force the binding of a cell to a particular view. The following files are used in the example:

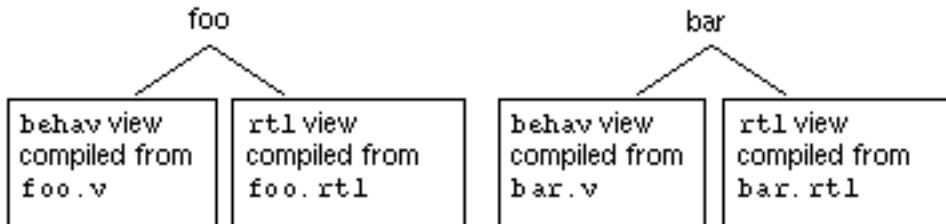
```
// hdl.var file
DEFINE VIEW_MAP (.v => behav,\n    .rtl => rtl,\n    .gate => gate )

DEFINE LIB_MAP (./designlib => designlib,\n    ./source => source,\n    + => worklib )
```

```
// File top.v
module top ();
    foo a();
    foo b();
    bar c();
endmodule
```

Diagram illustrating the binding of instances in the top module:

- Instance `a()`: You want to use the `rtl` view for these instances.
- Instance `b()`: You want to use the `behav` view for this instance.



```
;%# Compile top.v. The VIEW_MAP variable maps files with a .v extension to a
;%# view called behav. Compilation produces worklib.top:behav.
% ncvlog -messages top.v
ncvlog: v2.2.(p1): (c) Copyright 1995 - 1999 Cadence Design Systems, Inc.
file: top.v
module worklib.top:behav
    errors: 0, warnings: 0
```

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

```
# Compile foo.v. Compilation produces worklib.foo:behav.
% ncvlog -messages foo.v
ncvlog: v2.2.(p1): (c) Copyright 1995 - 1999 Cadence Design Systems, Inc.
file: foo.v
    module worklib.foo:behav
        errors: 0, warnings: 0

# Compile foo.rtl. The VIEW_MAP variable maps files with a .rtl extension to
# a view called rtl.Compilation produces worklib.foo:rtl.
% ncvlog -messages foo.rtl
ncvlog: v2.2.(p1): (c) Copyright 1995 - 1999 Cadence Design Systems, Inc.
file: foo.rtl
    module worklib.foo:rtl
        errors: 0, warnings: 0

# Compile bar.v. Compilation produces worklib.bar:behav.
% ncvlog -messages bar.v
ncvlog: v2.2.(p1): (c) Copyright 1995 - 1999 Cadence Design Systems, Inc.
file: bar.v
    module worklib.bar:behav
        errors: 0, warnings: 0

# Elaborate top. To resolve the first instance of foo, the library where the
# parent is located (worklib) is searched for a view that matches the view of
# the parent (behav). After resolving this instance, any following instances
# of foo receive the same binding.
% ncelab top -libverbose
ncelab: v2.2.(p1): (c) Copyright 1995 - 1999 Cadence Design Systems, Inc.
    Elaborating the design hierarchy:
Resolving module/udp 'foo' at 'top.a'.
    Caching library 'worklib' ..... Done
    library: 'worklib' views: 'behav' -> found
Resolved module/udp 'foo' at 'top.a' to 'worklib.foo:behav'.
Resolved module/udp 'foo' at 'top.b' to 'worklib.foo:behav'.
Resolving module/udp 'bar' at 'top.c'.
    library: 'worklib' views: 'behav' -> found
Resolved module/udp 'bar' at 'top.c' to 'worklib.bar:behav'.
Building instance overlay tables: ..... Done
    Loading native compiled code: ..... Done
    Building instance specific data structures.
```

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

```
Design hierarchy summary:
    Instances   Unique
    Modules:   4       3

Writing initial simulation snapshot: worklib.top:behav

;# Elaborate top. Use -binding to force binding of instances of foo to the rtl
;# view.

;# The first instance of foo is bound to the rtl view. Other instances of foo
;# use the same binding.

% ncelab top -binding foo:rtl -libverbose
ncelab: v2.2.(s2): (c) Copyright 1995 - 1999 Cadence Design Systems, Inc.

    Elaborating the design hierarchy:
Resolved module/udp 'foo' at 'top.a' to 'worklib.foo:rtl'.
Resolved module/udp 'foo' at 'top.b' to 'worklib.foo:rtl'.
Resolving module/udp 'bar' at 'top.c'.
    Caching library 'worklib' ..... Done
    library: 'worklib' views: 'behav' -> found
Resolved module/udp 'bar' at 'top.c' to 'worklib.bar:behav'.
    Building instance overlay tables: .....
    Loading native compiled code: .....
    Building instance specific data structures.

    Design hierarchy summary:
        Instances   Unique
        Modules:   4       3

Writing initial simulation snapshot: worklib.top:behav
```

## The ‘uselib Compiler Directive

The `uselib compiler directive overrides the default search mechanism and the -binding command-line option.

You can use special NC-Verilog extensions to `uselib or you can use the standard Verilog-XL syntax for this compiler directive. Verilog-XL syntax is translated to the NC-Verilog extensions.

### NC-Verilog Extensions to `uselib

Syntax:

```
`uselib lib = library_name
`uselib view = view_name
```

A library and a view can be specified with one directive as follows:

```
`uselib lib = library_name view = view_name
```

The library and the view can be specified in any order.

Each `uselib directive explicitly defines a library and/or view search that resolves the instances that follow it until the elaborator encounters another `uselib directive, which redefines the search. An empty `uselib directive or a `nouselib directive makes the preceding `uselib directives ineffective.

### Verilog-XL `uselib Syntax

The standard Verilog-XL syntax for `uselib is mapped to the NC-Verilog extensions as follows:

XL Usage	Is Mapped to:
<pre>`uselib dir = lib_directory_name</pre>	<pre>`uselib lib = library_name</pre> Based on the mapping of source directories to library names in the LIB_MAP variable.
<pre>`uselib file = lib_file_name</pre>	<pre>`uselib lib = library_name view = view_name</pre> Based on the mapping of source directories to library names in the LIB_MAP variable and on the mapping of file extensions to view names in the VIEW_MAP variable.
<pre>libext = file_extension</pre>	<pre>`uselib view = view_name</pre> Based on the mapping of file extensions to view names in the VIEW_MAP variable.

**Note:** If you do not specify both the library and the view in the `uselib directive, the missing mapping is taken from the hdl.var file. If the mapping of library or view cannot be made using the LIB\_MAP or VIEW\_MAP variables, all libraries defined in the cds.lib file are searched.

If you have a Verilog-XL design and run ncprep, an hdl.var file is created in the current directory. This hdl.var file will contain LIB\_MAP and VIEW\_MAP variables based on the

information in your `uselib compiler directives. See “[ncprep](#)” on page 867 for information on *ncprep*.

The following two examples show you how to use the `uselib compiler directive to force the binding of one particular instance of a module.

**Example 1:**

The following files are used in the first example. Notice that the `uselib directive in the file `top.v` specifies both the library and the view. The instance `a` of module `foo` will be bound to `source.foo:rtl`, as specified in the `uselib compiler directive.

```
// hdl.var file
DEFINE VIEW_MAP ( .v      => behav, \
                  .rtl    => rtl, \
                  .gate   => gate )

DEFINE LIB_MAP ( ./designlib => designlib, \
                  ./source     => source, \
                  +           => worklib )

// File source/top.v
module top ();
  `uselib lib=source view=rtl
  foo a();      // You want to use the rtl view in the library called source
  `uselib

  foo b();
  bar c();

endmodule

module foo
  source.foo:behav compiled from foo.v
  source.foo:rtl compiled from foo.rtl
  designlib.foo:rtl compiled with -work
endmodule

module bar
  source.bar:behav compiled from bar.v
  source.bar:rtl compiled from bar.rtl
endmodule
```

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

```
;# Compile all .v and .rtl files. The LIB_MAP variable maps source files in the
;# source directory to the library called source. The VIEW_MAP variable maps
;# files with a .v extension to a view called behav and .rtl files to a view
;# called rtl.

% ncvlog -messages source/foo.v
ncvlog: v03.40.(p1): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
file: source/foo.v
    module source.foo:behav
        errors: 0, warnings: 0

% ncvlog -messages source/bar.v
ncvlog: v03.40.(p1): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
file: source/bar.v
    module source.bar:behav
        errors: 0, warnings: 0

% ncvlog -messages source/foo.rtl
ncvlog: v03.40.(p1): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
file: source/foo.rtl
    module source.foo:rtl
        errors: 0, warnings: 0

% ncvlog -messages source/foo.rtl -work designlib
ncvlog: v03.40.(b001): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
file: source/foo.rtl
    module designlib.foo:rtl
        errors: 0, warnings: 0

% ncvlog -messages source/bar.rtl
ncvlog: v03.40.(p1): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
file: source/bar.rtl
    module source.bar:rtl
        errors: 0, warnings: 0

% ncvlog -messages source/top.v
ncvlog: v03.40.(p1): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
file: source/top.v
    module source.top:behav
        errors: 0, warnings: 0

;# Elaborate the top-level module.
;# Instance foo a is bound to source.foo.rtl.
;# Instances foo b and bar c are bound to the behav views using the default
;# binding mechanism.

% ncelab -messages -libverbose top
ncelab: v03.40.(b001): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
    Elaborating the design hierarchy:
```

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

```
Resolving module/udp 'foo' at 'top.a' ('uselib at ./source/top.v,6).
    Caching library 'source' ..... Done
    library: 'source' views: 'rtl' -> found
Resolved module/udp 'foo' at 'top.a' to 'source.foo:rtl' ('uselib at
./source/top.v,6).

Resolving module/udp 'foo' at 'top.b'.
    library: 'source' views: 'behav' -> found
Resolved module/udp 'foo' at 'top.b' to 'source.foo:behav'.

Resolving module/udp 'bar' at 'top.c'.
    library: 'source' views: 'behav' -> found
Resolved module/udp 'bar' at 'top.c' to 'source.bar:behav'.
    Building instance overlay tables: ..... Done
    Loading native compiled code: ..... Done
    Building instance specific data structures.

Design hierarchy summary:
    Instances   Unique
    Modules: 4        4
Writing initial simulation snapshot: source.top:behav
```

### **Example 2:**

The second example is identical to the first example, except that the 'uselib directive specifies only the view. Here is the file top.v:

```
module top ();

`uselib view=rtl
foo a();
`uselib

foo b();
bar c();

endmodule
```

The view specified in the compiler directive will be used. This will override any view specified with the -binding option. However, because the library is not specified, the libraries specified in the LIB\_MAP variable in the hdl.var file will be searched in order for an rtl view for module foo.

In this example, the library designlib will be searched first. Because there is a design unit called designlib.foo:rtl, this binding is used.

The following shows the output of the elaborator:

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

```
ncelab: v03.40.(b001): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.  
      Elaborating the design hierarchy:  
Resolving module/udp 'foo' at 'top.a' ('uselib at ./source/top.v,4).  
      Caching library 'designlib' ..... Done  
      library: 'designlib' views: 'rtl' -> found  
Resolved module/udp 'foo' at 'top.a' to 'designlib.foo:rtl' ('uselib at  
./source/top.v,4).  
Resolving module/udp 'foo' at 'top.b'.  
      Caching library 'source' ..... Done  
      library: 'source' views: 'behav' -> found  
Resolved module/udp 'foo' at 'top.b' to 'source.foo:behav'.  
Resolving module/udp 'bar' at 'top.c'.  
      library: 'source' views: 'behav' -> found  
Resolved module/udp 'bar' at 'top.c' to 'source.bar:behav'.  
      Building instance overlay tables: ..... Done  
      Loading native compiled code: ..... Done  
      Building instance specific data structures.  
Design hierarchy summary:  
      Instances Unique  
      Modules: 4 4  
Writing initial simulation snapshot: source.top:behav
```

## Enabling Read, Write, or Connectivity Access to Simulation Objects

By default, the elaborator marks all simulation objects in the design as having no read or write access, and disables access to connectivity (load and driver) information. Turning off these three forms of access allows the elaborator to perform a set of optimizations that can dramatically improve simulation performance.

The only exceptions to this default mode are objects that are used as arguments to user-defined system tasks or functions. These objects are automatically given read, write, and connectivity access. By default, no access is given to objects that are used as arguments to built-in system tasks or functions. Using a construct that does not have a value (a module instance, for example) as an argument has no effect on access capabilities.

Generating a snapshot with limited visibility into simulation constructs and running the simulation in *regression* mode has significant performance advantages. However, turning off access to the HDL data structures imposes the following limitation: You cannot access simulation objects from a point outside the HDL code, through Tcl commands or through PLI/VPI/VHPI.

This section describes:

- The limitations imposed by running in regression mode.
- How to turn on read, write, and connectivity access by using the following elaborator (*ncelab*) command-line options:
  - **-access**  
Use this option to specify the access capability for all objects in the design. See “[Using -access to Specify Read/Write/Connectivity Access](#)” on page 251.
  - **-afile**  
Use this option to include an access file, in which you specify the access capability for particular instances and portions of the design. See “[Using -afile to Include an Access File](#)” on page 253.  
See “[Generating an Access File](#)” on page 260 for information on how to automatically generate an access file.
- General guidelines for setting access control. See “[Guidelines for Access Control](#)” on page 261.

## Regression Mode and Tcl Commands

Many Tcl commands access and set values and probes on objects. Other commands provide load and driver information. With no read, write, or connectivity access to objects, these commands generate warning or error messages or they display output that does not include some objects or object values. For example:

- The `force` command prints an error if the object that is being forced does not have write access.
- The `deposit` command prints an error if the object to which a value is to be deposited does not have read/write access.
- The `value` command prints an error if any of the objects given as arguments do not have read access.
- The `probe` command prints an error if any of the objects given as arguments do not have read access. If the argument to this command is a scope, objects within that scope that do not have read access are excluded from the probe, and a warning message is printed. No waveforms are generated for these objects.
- The `drivers` command cannot display the drivers of a particular wire or register if the object does not have connectivity access.
- The `describe` command output does not include the value of an object if the object does not have read access.

## Regression Mode and PLI/VPI/VHPI Applications

If you run in the default regression mode, PLI/VPI/VHPI applications:

- Cannot get the values of objects tagged as having no read access.

If the value of an object that does not have read access is requested, the PLI interface returns the value as a strong X in the requested format. If the requested format is decimal, integer, or time, the value is 0. If the format is in the form of a double, the return value is 0.0.

If PLI 1.0 routines are being used, the ACC flag `acc_error_flag` is set to non-zero. This error can be detected in the VPI interface by calling `vpi_chk_error()` after a call to `vpi_get_value()` or by registering a callback for `cbPLIError`.

- Cannot put values to objects tagged as having no write access.

Trying to put a value to a non-writable object is ignored and results in an error. If an event handle is requested from the `vpi_put_value()` routine, the return value is NULL.

If PLI 1.0 routines are being used, the ACC flag `acc_error_flag` is set to non-zero. This error can be detected in the VPI interface by calling `vpi_chk_error()` after a call to `vpi_put_value()` or by registering a callback for `cbPLIError`.

- Cannot place value change callbacks on objects tagged as having no read access.

Requesting a value change callback on a non-readable object is ignored and results in an error. Calling `acc_vcl_add()` on the object results in the ACC flag `acc_error_flag` being set to non-zero. Calling `vpi_register_cb()` on the object results in a NULL being returned. This error can be detected using `vpi_chk_error()` immediately after the call to `vpi_register_cb()` or by registering a callback for `cbPLIError`.

- Cannot place callbacks on objects for force and release if the objects are tagged as having no read access.
- Cannot scan for loads or drivers of an object without connectivity access.  
If the object does not have read connectivity access, scanning for loads results in an error. Scanning for drivers on an object that does not have read access also generates an error.
- The simulated net for a net without read access is the original net.

### Testing for the Visibility of an Object

The PLI 1.0, VPI, and VHPI interfaces allow an application to test for the accessibility of an object before working on it. The following macros are defined in `vxl_acc_user.h`:

- `accWriteAccess`
- `accReadAccess`
- `accConnectivityAccess`

You can use these macros with `acc_object_of_type()` and `acc_object_in_typelist()`.

The following properties are included in `vpi_user_cds.h`:

- `vpiWriteAccess`
- `vpiReadAccess`
- `vpiConnectivityAccess`

These Boolean properties can be used with the `vpi_get()` routine. They return TRUE if the object is read/write accessible and return FALSE otherwise.

These properties are accessible from any object that can have a value. In addition, the calls `vpi_get(vpiWriteAccess, NULL)` and `vpi_get(vpiReadAccess, NULL)` return FALSE if any object in the design has limited visibility. In Verilog-XL, these properties are accessible and always return TRUE.

The property `vhpiAccessP` is defined in `vhpi_user.h`. You can use this property to test the visibility of VHPI objects: `vhpi_get(vhpiAccessP, objHandle)`.

### Controlling the Display of PLI Error and Warning Messages

By default, the simulator displays all warning and error messages that are generated when an error is detected due to a PLI read, write, or connectivity access violation. You can suppress the display of these access violation messages by using the `-plinoptwarn` command-line option when you invoke the simulator (`ncsim`). If you use this option, a warning message is displayed only once when the first violation is detected. The message is displayed again if an access violation is detected after a reset or a restart.

### Regression Mode and the SimVision Debug Environment

SimControl and its debug tools cannot display the value of objects that do not have read access. The current value for an object without read access is shown as three question marks (???). Trying to execute commands such as *Show—Value*, *Set—Breakpoint—Object*, or *Set—Probe* on objects without read access results in an error message.

The *Set—Force* command generates an error message for objects that do not have write access.

Connectivity access affects the Signal Flow Browser. For signals that do not have connectivity access, a warning message is displayed and no action is taken. In other cases, an object may have connectivity access, but some drivers have been removed or have been optimized. In these cases, a warning is displayed telling you that the signal has no drivers.

### Using -access to Specify Read/Write/Connectivity Access

Use the `ncelab` elaborator `-access` option to turn on the different kinds of access. You can use this option to specify the default read, write, or connectivity access to all simulation objects in the design.

By default, objects do not have read, write, or connectivity access. In other words, the default is -access -r-w-c.

Objects that are given write access are also given read access. Objects that are given connectivity access are given write and read access.

Syntax:

```
-access [+/-] access_spec
```

The *access\_spec* argument can be:

- r (read access)
- w (write access)
- c (connectivity access)
- Any combination of the three access types

Use the plus sign to turn on the specified access. This is the default if no plus or minus sign is used with -access. Use the minus sign to turn off the specified access.

The + and - options apply to all subsequent r, w, or c specifications until the next + or -.

Examples:

- Read access only:

```
-access +r (same as -access r)
```

- Write access:

```
-access +w (same as -access w)
```

Objects given write access are also given read access.

- Read/Write access:

```
-access +rw (same as -access +r+w and -access rw)
```

- Connectivity access:

```
-access +c (same as -access c)
```

Objects given connectivity access are also given write and read access.

You can also use multiple -access options. For example,

```
-access +r -access +c
```

**Note:** Objects that are given connectivity access are given write access, and objects that are given write access are given read access. With the following set of options, all objects are given connectivity access and, therefore, write and read access.

```
-access +c -access -rw
```

Use the following general rules when deciding what kind of access you want to specify:

- Read access is required if you want to probe objects in the design and generate an SHM, VCD, or EVCD database. This lets you use SimVision Waveform Viewer to view waveforms, Comparescan to compare databases, and most Tcl commands and SimControl features. Read access is also required for getting signal values with, for example, the `value` or `describe` command, or `vpi_get_value()`.
- Write access is required if you want to deposit values using, for example, the `force` or `deposit` command, or `vpi_put_value()`.
- Connectivity access is required in order to show load or driver information. This kind of access is required, for example, by the `drivers` command and by the Signal Flow Browser.
- If you know that you will require some debug capability, but are not sure what kind of access to specify, turn on all access with `-access +rwc`.

## Using `-afile` to Include an Access File

An access file is a text file that lets you specify the type of access that you want for particular instances and portions of the design. This section tells you how to include the access file when you invoke the elaborator and how to write an access file.

### Including the Access File

To include an access file, use the `-afile access_file` option when you invoke the elaborator. If you are running `ncverilog`, use the `+ncafile+access_file` command-line option.

```
% ncelab -afile access_file [lib.]cell[:view]
```

For example,

```
% ncelab -afile afle.af worklib.top  
(% ncverilog +ncafile+afle.af source.v)
```

You can use more than one `-afile` option to include multiple access files. If you use multiple `-afile` options, the contents of the different files are combined before the access is actually set. For example,

```
% ncelab -afile afile1.af -afile afile2.af worklib.top
```

### Writing an Access File

An access file consists of a set of lines, each of which specifies the desired access capability for instances in the design, hierarchical portions of the design, or value constructs used as arguments to user-defined system tasks or functions.

Each line in the access file begins with a keyword. You can enter keywords in uppercase or lowercase. Each line must be terminated by a carriage return.

Some keywords require a hierarchical path argument. The syntax for specifying hierarchical paths is language-neutral. You can use either the VHDL path element separator (a colon), or the Verilog path element separator (a period) to separate the path elements. A path that starts at the VHDL top-level must begin with a colon.

Two wildcard extensions can be used in hierarchical path specifications:

- An asterisk ( `*` ) matches any instance in the current scope.
- Three dots ( `...` ) used as a suffix matches any instance in the hierarchy below the current scope.

The access specifications in the access file are similar to those used with the `-access` command-line option. Use a plus sign to turn on the specified access and a minus sign to turn off the specified access. Use `r`, `w`, and `c` for read, write, and connectivity access, respectively. Objects that are given connectivity or write access are also given read access.

However, in an access file you can also specify `+d`. This specification gives all drivers of a net read access if the net has connectivity access. This provides a convenient way to provide access so that an application can scan for all drivers and read their values.

**Note:** With the `-access` option, the plus sign is the default if you do not specify a plus or minus sign. In an access file, you must use the plus sign to turn on access.

You can also include comments in the file.

- Begin one-line comments with two slashes ( `//` ).
- Begin multiple-line comments with `/*` and end the comment with `*/`.

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

Here is a simple example access file:

```
// Read access for all instances in the 2nd levels of hierarchy
PATH *.*      +r-wc

/* Read and write access for all instances of sub that are
two levels under top */
PATH top.*.sub  +rw-c

//Read access, but no write access, for all instances below top.sub
PATH top.sub... +r-wc
```

Objects derive their access from an exact match. If there is no exact match for an object, the access is derived from the closest matching wildcard. If an object cannot be matched with a wildcard, the default access is used. For example, if you have the following two lines in an access file, top.u1 and all instances inside u1 have full access enabled because the first line specifies the more complete path.

```
PATH top.u1.* +rwc
PATH top.*.* -rwc
```

### Access File Syntax

The syntax of the access file is as follows:

```
Access_file ::= Line
Line ::= 
    DEFAULT Access
    | BASENAME [Path] [Access]
    | PATH Path Access
    | CELLINST Access
    | CELLLIB Cell Access
    | $UDTF Stfname Access
    | INCLUDE File
    | Comments

Path ::= 
    Name
    | Path.Name
    | Path...
    | Path...Key
    | Path.Key

Cell ::= A name in Lib.Cell:View format

Name ::= Legal Verilog or VHDL name
```

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

```
Stfname ::=  
    Name of a user-defined system task or function  
    | *  
  
File ::= Path to an access file to include  
  
Key ::=  
    <REG>  
    | <WIRE>  
    | <INTEGER>  
    | <TIME>  
    | <REAL>  
    | <PRIMITIVE>  
    | <ASSIGN>  
    | <EVENT>  
    | <PORT>  
    | <PORTIN>  
    | <PORTOUT>  
    | <PORTINOUT>  
    | <SIGNAL>  
    | <VARIABLE>  
  
Access ::= Modifier Capability  
Modifier ::= + | -  
Capability ::= R | W | C | D | RW | RC | WC | RWC | CD  
  
Comments ::=  
    // text  
    | /* text */
```

The keywords that you can use in an access file are:

■ **DEFAULT Access**

Specifies the default access for all instances. This overrides any access that you specify with the `-access` option.

**Example:**

```
DEFAULT +r-wc
```

■ **BASENAME [Path] [Access]**

Specifies the starting point for the path in all subsequent PATH statements. The specified path cannot contain any wildcard characters. If you do not specify a path, the BASENAME is set to null.

**Example:**

```
BASENAME top.ul.foo          // Start all subsequent paths with top.ul.foo
PATH bar +rwc                // top.ul.foo.bar has +rwc
PATH u3 -rwc                 // top.ul.foo.u3 has -rwc
BASENAME                      // Remove previous basename specification
PATH top.u3 +rwc              // top.u3 has +rwc
```

■ **PATH Path Access**

Specifies the access for all instances and constructs that match the path specification.

If a path ends with an object name, the named object gets the specified access. The following example turns on read and connectivity access for `top.foo.io`. The `+d` specifies that all drivers of a net with connectivity access will have read access:

```
PATH top.foo.io +rcd-w
```

Objects from named blocks or HDL tasks and functions get their access from the containing scope if they do not have an exact match.

You can use wildcard characters in the hierarchical path argument. The two wildcard characters are:

- ❑ An asterisk ( `*` ) matches any instance at the current scope.
- ❑ Three dots ( `...` ) used as a suffix matches any instance in the hierarchy below the current scope.

The following example turns read and write access on for the top two levels of the design:

```
PATH * +rw-c           // Read, write access for top level
PATH *.* +rw-c         // Read, write access for second level
PATH ... -rwc          // No access to objects below the second level
```

In addition to normal Verilog or VHDL hierarchical paths, such as `top.ul.foo`, and hierarchical names using wildcard characters, there are several keys that let you specify access for particular constructs. If the path ends with a key, all objects of a class that matches the key get the specified access. These keys are:

- ❑ `<REG>` (Matches Verilog registers not declared as `integer`, `real`, or `time`)
- ❑ `<INTEGER>, <REAL>, <TIME>` (Matches the corresponding type of register declaration)

- ❑ <WIRE> (Matches Verilog wires)
- ❑ <SIGNAL> (Matches VHDL signals and ports)
- ❑ <VARIABLE> (Matches VHDL variables)
- ❑ <PORT> (Matches all Verilog wires and registers declared as module ports, and all VHDL ports)
- ❑ <PORTIN>, <PORTOUT>, <PORTINOUT> (Match only ports of the corresponding mode)
- ❑ <PRIMITIVE> (Matches Verilog instances of primitives)
- ❑ <ASSIGN> (Matches Verilog blocking assignment statements)
- ❑ <EVENT> (Matches Verilog named events)

**Examples:**

```
PATH top.<WIRE> +rw-c      // Read, write access for wires in second level  
PATH top.sub...<REG> +rw-c /* Read, write access for regs in instances below  
                           top.sub */  
PATH top.sub.foo.<PRIMITIVE> +rw-c // Read, write access for primitives in foo
```

If an object can be matched by either a <PORT\*> or <WIRE> key, the <PORT\*> key is used.

■ **CELLINST Access**

Specifies the access for all instances that are tagged as cells and their subhierarchy. Instances are tagged as cells either with the `celldefine compiler directive or by using the -y or -v options in *ncverilog*.

The access that you specify with CELLINST overrides all wildcard paths that match into a cell instance. However, an object in a cell instance matched by an exact path is annotated using the access from the exact path.

The following example specifies that all cells in the design can be fully optimized, while the upper levels have full debug access:

```
CELLINST -rwc      // No access to objects in cell instances  
PATH ... +rwc      // Enable full access to objects above cells
```

■ **CELLLIB Cell Access**

Specifies access using a lib.cell:view format.

**Example:**

```
CELLLIB worklib.m16:module +rw-c
```

You can use the \* wildcard character for any part of the lib.cell:view specification.

**Examples:**

```
CELLLIB worklib -rwc  
CELLLIB worklib.* +rwc  
CELLLIB worklib.m16:* +rwc  
CELLLIB *.m16 +rwc
```

In the following example access file, CELLLIB is used to turn off access to all objects in the library asic:

```
// Full access to all objects  
PATH ... +rwc  
// No access to objects in the library asic  
CELLLIB asic -rwc  
/* Read access to register r1 in the instance top.ul.as01, an instance of a  
part in the library asic */  
PATH top.ul.as01.r1 +r-wc
```

■ **\$UDTF *Stfname* Access**

Specifies the access for value constructs used as arguments to a user-defined system task or function. By default, these constructs are given full access (+rwc). Use this keyword to turn off different kinds of access.

**Example:**

```
$UDTF $mytask +r-wc
```

You can use \* to specify all user-defined system tasks and functions.

```
$UDTF * +r-wc
```

**Note:** The default access for constructs used as arguments to built-in system tasks and functions is -rwc. You cannot use \$UDTF to modify this access.

■ **INCLUDE *File***

Include the contents of the specified access file.

## **Warning Messages**

There are several conditions that result in warning messages. These include:

- If an access mode for an object is both enabled and disabled, access is enabled, and a warning is issued. For example, in the following access file, top.ul.u3 gets read access.

```
PATH top.u1.u3 +r-wc
...
...
PATH top.u1.u3 -rwc
```

- A warning is issued if an object is named in an access file, but is not used in the elaborated design. For example, a warning is generated if you have the following line in the access file, but `top.u1.u3` is not used in the design.

```
PATH top.u1.u3 +r-wc
```

No warning is generated if wildcards are used. For example, if you have the following line in the access file, no warning is issued if `b` is not found in the design.

```
PATH top.*.b +r-wc
```

## Generating an Access File

If you know that your simulation runs require the same types of access on the same objects, you can automatically generate an access file. To do this, include the `-genafile` option when you invoke the elaborator. When you simulate, the objects that are accessed by Tcl commands or by a PLI application are monitored along with the types of access required for each object, and when you exit the simulation, an access file is created with the specified filename.

Example:

```
% ncelab -genafile access.txt top
```

You can then include this access file in subsequent runs with the `-afile` option.

```
% ncelab -afile access.txt top
```

If you use `-genafile`, any request for access reduction with the `-access` or `-afile` command-line options is ignored. The `describe` command (*Show—Description* on the SimControl window) does not affect the specifications inserted into the access file.

**Note:** If you are running `ncverilog`, use the `+ncgenafile+access_filename` option to generate an access file. Then use `+ncafle+access_file` to include the access file. If you want to use the reinvoke simulation feature on the SimControl window, select *Options—Preferences* and make sure that the *Prompt before reinvoke* option is set. When you then select *File—Reinvoke Simulation*, the Reinvoke form appears and you can edit the text field that contains your original `ncverilog` command-line options to change `+ncgenafile+` to `+ncafle+`.

## Guidelines for Access Control

You can trade off simulation performance for debugability using the access control mechanism in the NC-Verilog simulator. By default, the NC-Verilog simulator runs in maximum performance mode. This means that you will have minimal debug access if you run in the default mode. If you are going to use Tcl interactive commands or PLI/VPI/VHPI routines to debug and/or analyze the design, some amount of access is required.

This section provides general guidelines for access control. There are three sections:

- [General Access Control Guidelines](#)
- [Access Requirements for Tcl Interactive Commands](#)
- [Access Requirements for PLI/VPI/VHPI Functions and Callbacks](#)

### General Access Control Guidelines

No special access is required for viewing the hierarchy or for finding the names of objects (nets, regs, variables, scopes, and so on) in the design.

Read access (+R) is required for probing nets, regs, and variables (including setting PLI callbacks) and getting the value of these objects.

Write access (+W) is required to interactively set the value of simulation objects (depositing or forcing variables). Write access automatically provides read access.

**Note:** With the exception of system tasks and functions, HDL constructs, such as a force statement, do not require any special access.

Connectivity access (+C) is required to get driver and load information about a specific net, reg, or other variable. Connectivity access automatically provides write and read access.

Compiling with the -linedebug option (`ncvlog -linedebug`) is required for setting breakpoints at source lines or for applying statement callbacks. Using this option automatically provides read, write, and connectivity access.

Elaborating with the -anno\_simtime option (`ncelab -anno_simtime`) is required if you want to use PLI/VPI routines that modify delays at simulation time. Using this option automatically provides read, write, and connectivity access.

*ncverilog* provides three ways to change the default access of a design.

- +debug sets read access (that is, +R).

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

- `+ncaccess+` lets you to set the access you need for the design (for example, `+ncaccess+r` or `+ncaccess+wc`).
- `+ncafile+` lets you specify an access file, in which you set the access for particular instances or portions of a design.

These general guidelines can be summarized as follows: specify only the type(s) of access required for your debugging purposes, and try to set access controls on specific scopes, nets, regs, ports, and so on, by using an access file.

For example, if the only reason you need access is to save waveform data, use `ncelab -access +R(ncverilog +ncaccess+R)`. If possible, use an access file to set read access only on the scopes or individual variables that you want to probe. The following access file, for example, can be used to save waveform data for ports only. The `DEFAULT` keyword specifies the default access for all instances, and the second statement specifies read access for ports.

```
DEFAULT -rwc
PATH ...<PORT> +r-wc
```

The following access file can be used to save waveform data for all nets and regs only:

```
DEFAULT -rwc
PATH ...<WIRE> +r-wc
PATH ...<REG> +r-wc
```

The following access file can be used if you want to save waveform data for ports only and perform scan tests by writing to scanflops tagged as cells with `celldefine:

```
DEFAULT -rwc
PATH ...<PORT> +r-wc
CELLINST worklib.dff_scan.module +rw-c
```

### Access Requirements for Tcl Interactive Commands

The following table lists Tcl commands and the type of access they require:

---

Tcl Command	Access Requirement
alias	None
call	Depends on the C function or PLI/VPI/VHPI routine
database	None
deposit	+W

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

Tcl Command	Access Requirement
describe	None (+R to see object values)
drivers	+C
finish	None
force	+W
help	None
probe	
[-create]	+R (for all desired objects)
-delete	None
-disable	None
-enable	None
-show	None
release	None
reset	None
restart	None
run	None
save	None
scope	
-describe	None (+R to see object values)
-names	None
-sort	None
-drivers	+C
-list	None
[-set]	None
-show	None
status	None

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

Tcl Command	Access Requirement
stop	
[-create]	
-condition	+R (if condition include simulation object values)
-continue	None
-delbreak	None
-execute	None
-if	+R (if condition must include object values)
-line	-linedebug ( <i>ncvlog</i> option for desired file)
-name	None
-object	+R
-skip	None
-time	None
-delete	None
-disable	None
-enable	None
-show	None
time	None
value	+R
version	None

---

## Access Requirements for PLI/VPI/VHPI Functions and Callbacks

<b>PLI 1.0 Function</b>	<b>Access Requirement</b>
acc_vcl_add()	+R
acc_fetch_value()	+R
acc_fetch_paramval()	+R
acc_fetch_paramval_mtm()	+R
acc_set_value()	+W
acc_next_driver()	+C
acc_next_load()	+C
acc_append_delays()	-anno_simtime
acc_fetch_delays()	-anno_simtime
acc_replace_delays()	-anno_simtime
acc_append_pulsere()	-anno_simtime
acc_fetch_pulsere()	-anno_simtime
acc_replace_pulsere()	-anno_simtime
acc_set_pulsere()	-anno_simtime
misc_tf reason codes reason_paramvc	+R

<b>VPI Function</b>	<b>Access Requirement</b>
vpi_get_value()	+R
vpi_put_value()	+W
vpi_get_delays	-anno_simtime
vpi_iterate(vpiDriver)	+C
vpi_iterate(vpiLoad)	+C
vpi_put_delays	-anno_simtime

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

VPI Function	Access Requirement
vpi_register_cb()	
cbValueChange	+R
cbForce/cbRelease	+R (on the expression)
cbAssign/cbDeassign	+R (on the register being assigned)
cbStmt	-linedebug (on desired module)

VHPI Function	Access Requirement
vhpi_get_value()	+R
vhpi_put_value()	+W
vhpi_iterator()	
vhpiContributors	+C
vhpiDrivers	+C
vhpi_register_cb()	
vhpiCbValueChange	+R
cbStm	-linedebug

## Disabling Timing in Selected Portions of a Design

You can turn off timing in specific parts of a Verilog design by using a timing file, which you specify on the command line with the `-tfile` option. For example,

```
% ncelab -tfile myfile.tfile worklib.top:module
```

If you are running NC-Verilog in single-step invocation mode, use the `+nctfile+` option. For example,

```
% ncverilog +nctfile+myfile.tfile source_files
```

If you are annotating with an SDF file, the design is annotated using the information in the SDF file, and then the timing constructs that you specify in the timing file are removed from the specified instances.

Using a timing file does not cause any new SDF warnings or remove any timing warnings that you would get without a timing file. There is one exception to this: The connectivity test for register driven interconnect delays happens much later than the normal interconnect delays. Any warning that may have existed for that form of the interconnect will not be generated if that interconnect has been removed by a timing file.

### Writing a Timing File

A timing file consists of a set of lines, each of which specifies whether or not you want timing for specific instances in the design or for hierarchical portions of the design. Each line begins with a keyword. You can enter the keywords in uppercase or in lowercase. Each line must be terminated by a carriage return.

Some keywords require a hierarchical path name argument. Two wildcard extensions can be used in hierarchical path specifications:

- An asterisk ( `*` ) matches any instance at the current scope.
- Three dots ( `...` ) used as a suffix matches any instance in the hierarchy below the current scope.

The timing specifications in the file are as follows:

- `- | + iopath`—Removes module path delays.
- `- | + prim`—Sets any primitive delay within the specified instance(s) to 0.
- `- | + port`—Removes any port delays at the specified instance(s) or any interconnects whose destination is contained by the instance. Interconnect sources are not affected by the `-port` construct.

- - | + tcheck—Removes all timing checks from the instance(s).
- - | + timing—This is an alias for the four specifications shown above.

You can also include comments in the file. Begin one-line comments with two slashes ( // ). Begin multiple-line comments with /\* and end the comment with \*/.

Here is a simple example timing file:

```
// Disable timing checks in top.foo
PATH top.foo -tcheck
// Disable timing checks in all scopes below top.foo
PATH top.foo... -tcheck
// Enable timing checks in top.foo.bar
PATH top.foo.bar +tcheck
// Disable timing checks for all objects in the library mylib
CELLLIB mylib -tcheck
// No module path delays for all instances in the 2nd levels of hierarchy
PATH *.* -iopath
```

The keywords that you can use in a timing file are:

- DEFAULT *timing\_spec*

Specifies the default timing behavior for all instances.

Example:

```
DEFAULT -timing
```

- BASENAME [*path*] [*timing\_spec*]

Specifies the starting point for the path in all subsequent PATH statements. The specified path cannot contain any wildcard characters. If you do not specify a path, the BASENAME is set to null.

Example:

```
BASENAME top.counter          // Start all subsequent paths with top.counter
PATH U1 -iopath               // No module path delays for top.counter.U1
PATH U2 -port                 // No port delays for top.counter.U2
BASENAME                      // Remove previous basename specification
PATH top.counter.U3 -tcheck   // No timing checks for top.counter.U3
```

- PATH *path* *timing\_spec*

Specifies whether timing is on or off for all instances that match the path specification. You can use wildcard characters in the hierarchical path name argument. The two wildcard characters are:

- ❑ An asterisk ( \* ) matches any instance at the current scope.
- ❑ Three dots ( . . . ) used as a suffix matches any instance in the hierarchy below the current scope.

The following example turns off timing checks for the top two levels of the design and module path delays for instances below the second level:

```
PATH * -tcheck // No timing checks for top level  
PATH *.* -tcheck // No timing checks for second level  
PATH ... -iopath // No module path delays for instances below the second level
```

■ **CELLINST *timing\_spec***

Specifies whether timing is on or off for all instances that are tagged as cells and their subhierarchy. Instances are tagged as cells either with the `'celldefine` compiler directive or by using the `-y` or `-v` options in *ncverilog*.

The timing behavior that you specify with **CELLINST** overrides all wildcard paths that match into a cell instance. However, an object in a cell instance matched by an exact path is annotated using the access from the exact path.

In the following example, the **PATH** statement turns off timing for all instances. The **CELLINST** statement turns on timing for all instances marked as cells.

```
PATH ... -timing // Turn off timing for all instances  
CELLINST +timing // Turn on timing for all cell instances
```

■ **CELLLIB *lib.cell:view timing\_spec***

Specifies whether timing is on or off, using a lib.cell:view format.

Example:

```
CELLLIB worklib.m16:module -timing
```

You can use the \* wildcard character for any part of the lib.cell:view specification.

Examples:

```
CELLLIB worklib -timing  
CELLLIB worklib.* -timing  
CELLLIB worklib.m16:* -timing
```

■ **INCLUDE *timing\_file***

Include the contents of the specified timing file.

## Selecting a Delay Mode

Delay modes let you alter the delay values specified in your models by using command-line options and compiler directives. 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 on a module basis. If you assign a specific delay mode to a module, all instances of that module simulate in that mode. The delay mode of each module is determined at elaboration time and cannot be altered dynamically.

**Note:** 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.

### Delay Modes

The following sections describe the four delay modes that you can explicitly select and the default mode in effect if no delay mode is selected.

#### Unit Delay Mode

In unit delay mode, the NC-Verilog simulator 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 273).

To override the effect of the unit delay mode for specific delays, you can use:

- PLI access routines (see the *PLI 1.0 User Guide and Reference* and the *VPI User Guide and Reference* for more information)
- the `DelayOverride$ specparam` in a `specify` block. See “[DelayOverride\\$ specparam](#)” on page 275.

#### 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.

To override the effect of the zero delay mode for specific delays, you can use:

- PLI access routines (see the *PLI 1.0 User Guide and Reference* and the *VPI User Guide and Reference* for more information)
- the `DelayOverride$ specparam` in a specify block. See “[DelayOverride\\$ specparam](#)” on page 275.

### 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, the NC-Verilog simulator 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*). The NC-Verilog simulator ignores the `DelayOverride$ specparam` in the distributed delay mode.

### Path Delay Mode

In this mode, the NC-Verilog simulator 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 (with the exception of `trireg` charge decay times) are set to zero. In path delay mode, `trireg` 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 `DelayOverride$ specparam` or with PLI access routines (see the *PLI 1.0 User Guide and 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. See “[DelayOverride\\$ specparam](#)” on page 275.

When path delay mode is selected, modules that contain *no* module path delays simulate in distributed delay mode.

### Default Delay Mode

If you do not specify a delay mode, the model simulates in the default mode. Delays simulate as specified in the model’s source description files. 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 detailed timing checks. 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 or the distributed delays at elaboration time. This feature allows you to use the same source description with multiple tools and then to select the appropriate delay mode when using the sources with the NC-Verilog simulator. You can set the delay mode for the NC-Verilog simulator by placing a compiler directive for either the distributed or path mode in the module source description file or by specifying a global delay mode at elaboration time.

## Setting a Delay Mode

There are two ways to set a delay mode:

- Use command-line options when you invoke the elaborator to set a global delay mode.
- Use compiler directives in the source file to set delay modes specific to particular modules.

The order of precedence in delay mode selection from highest to lowest is as follows:

1. Command-line option
2. Compiler directives
3. Default — no delay mode

### Command-Line Options

There are four command-line options you can use to set a global delay mode. If you use more than one option, the elaborator issues a warning and selects the mode with the highest precedence. The options are listed in the following table from highest to lowest precedence:

-delay_mode path	The design simulates in path delay mode—except for modules with no module path delays.
-delay_mode distributed	The design simulates in distributed delay mode.
-delay_mode unit	The design simulates in unit delay mode.
-delay_mode zero	Modules simulate in zero delay mode.

## 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:

- `delay\_mode\_path
- `delay\_mode\_distributed
- `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. Delay modes specified with a compiler directive remain active across file boundaries. 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 can override all compiler directives by using the command-line options.

## Timescales and Simulation Time Units

When working with delay modes, you should consider the way delay modes use timescales and simulation time units. When you select the unit delay mode, 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);
```

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

When a model uses timescales, the 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.) For example, the gate shown above might be controlled by the following timescale directive:

```
'timescale 1 us/1 ns
```

This directive causes the simulation delay value for the nand gate g1 to be five microseconds. Elsewhere in the model, you might have the following timescale directive, which 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);
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
- the `DelayOverride$ specparam` in a specify block

### PLI Access Routines and Delays

You can use a PLI 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 PLI routines that set delay values are the following:

- `acc_append_delays`
- `acc_replace_delays`

An application can use the `acc_fetch_delay_mode` access routine to retrieve delay mode information.

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.

### **DelayOverride\$ specparam**

Modules frequently need distributed delays on sequential elements to prevent race conditions. Sometimes such a module also needs path delays. Use the `DelayOverride$`

specparam to ensure that these essential delays are not overridden in path, unit, or zero delay modes.

The DelayOverride\$ specparam lets you 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, the NC-Verilog simulator 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 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 example shows how to use the DelayOverride\$ specparam:

```
module
  ...
  nand #5 g1 (q, qbar, preset) ;
  ...
  specify
    ...
    specparam DelayOverride$g1= 5;
    ...
  endspecify
  ...
endmodule
```

When using DelayOverride\$, the delay override value is measured in simulation time units—that is, the module's timescale is ignored.

**Note:** In Verilog-XL, when you use `+delay_mode_path` with the `DelayOverride$specparam`, some gates are assigned zero delay and some gates are assigned the override value. In the NC-Verilog simulator, all gates are assigned the override value.

## 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 option.

```
'delay_mode_distributed          // compiler directive controls all instances
                                // of ffnnand
module ffnnand (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 when you select one of the global delay modes:

unit delay	Gate <code>g1</code> is assigned a delay value of five simulation time units because the <code>specparam DelayOverride\$g1</code> overrides the unit delay mode; gate <code>g2</code> keeps its zero delay because unit delay mode affects only non-zero delays.
zero delay	Gate <code>g1</code> gets a delay of five simulation time units, as specified by the <code>specparam DelayOverride\$g1</code> .
distributed delay	A global distributed delay mode has the same effect on this module as no global delay mode because the compiler directive selects distributed mode. In either case, <code>g1</code> has a delay of one timescale unit because the distributed delay is used (the <code>specparam</code> and module path specification are both ignored).

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>);
```

## 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.

	<b>Unit</b>	<b>Zero</b>	<b>Distributed</b>	<b>Path**</b>	<b>Default</b>
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 <code>DelayOverride\$</code>	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

---

## Setting Pulse Controls

This section discusses:

- How to set global pulse control for module path delays and/or interconnect delays using command-line options when you elaborate the design. See “[Global Pulse Control](#)” on page 281.
- How to set module path pulse control for a specific module or for particular paths within modules using the `PATHPULSE$` specparam. See “[Pulse Control for Specific Modules and Module Paths](#)” on page 283.
- How to use the *On-Event* and *On-Detect* pulse filters. See “[Pulse Filtering Style](#)” on page 285.

This section does not discuss SDF annotation. See “[PATHPULSE Keyword](#)” on page 1023 and “[PATHPULSEPERCENT Keyword](#)” on page 1024 for more information on SDF annotation of pulse control limits.

### Overview

In the NC-Verilog simulator, both module path delays and interconnect delays are simulated as transport delays by default. There is no command-line option to enable the transport delay algorithm. You must, however, set pulse control limits to see transport delay behavior. If you do not set pulse control limits, the limits are set equal to the delay by default, and no pulses having a shorter duration than the delay will pass through. That is, if you do not set pulse control limits, module path delays and interconnect delays are simulated as transport delays, but the results look as if the delays are being simulated as inertial delays.

See [Chapter 16, “Interconnect and Module Path Delays.”](#) for details on interconnect and module path delays.

Full pulse control is available for both types of delays. You can:

- Set global pulse control for both module path delays and interconnect delays.
- Set global pulse limits for module path delays and interconnect delays separately in the same simulation.
- Narrow the scope of module path pulse control to a specific module or to particular paths within modules using the `PATHPULSE$` specparam.
- Specify whether you want to use *On-Event* or *On-Detect* pulse filtering.

## Global Pulse Control

Use the `-pulse_r` and the `-pulse_e` options when you invoke the elaborator to set global pulse control. These options set global pulse limits for both module path delays and interconnect delays.

If you want to set pulse control for module path delays and interconnect delays separately in the same simulation, use the following two sets of options:

- `-pulse_r` and `-pulse_e` to set limits for path delays
- `-pulse_int_r` and `-pulse_int_e` to set limits for interconnect delays

By setting a global pulse control, you tell the simulator to take one of the following actions:

- Reject the output pulse (the state of the output is unaffected).
- Let the output pulse through (the state of the output reflects the pulse).
- Filter the output pulse to the error state. This generates a warning message and then maps to the `x` state.

**Note:** Pulse widths are measured at the output, not at the input.

The action that the simulator takes depends on the delay value and a window of acceptance. The simulator calculates the window of acceptance from the following two values that you supply as arguments to the options. Both arguments are percents of the delay.

- `reject_percent`
- `error_percent`

Syntax:

```
-pulse_r reject_percent -pulse_e error_percent [Lib.]Cell[:View]
```

Example:

```
% ncelab -pulse_r 50 -pulse_e 80 top_mod
```

The calculation of the limits is as follows:

```
reject_limit = (reject_percent / 100) * delay  
error_limit = (error_percent / 100) * delay
```

For example, the command line shown above specifies a `reject_percent` of 50% and an `error_percent` of 80%. This means that, for a module path delay of 50 time units, the reject limit is 25 time units (50% of 50 time units) and the error limit is 40 time units (80% of 50 time units).

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

Using the reject limit and error limit calculations, the simulator acts on pulses according to the following rules:

- Reject if  $0 \leq \text{pulse} < (\text{reject limit})$ .
- Set to error if  $\text{reject limit} \leq \text{pulse} < (\text{error limit})$ .
- Pass if  $\text{pulse} \geq \text{error limit}$ .

Therefore, in our example, in which the module path delay is 50 time units:

Output Pulse Width	Result
0 - 24	Reject
25 - 39	Set to error
40+	Pass

To generate an error whenever a module path pulse is less than the module path delay, use the following values:

```
-pulse_r 0 -pulse_e 100
```

The values of *reject\_percent* and *error\_percent* must fall between 0 and 100, with *reject\_percent*  $\leq$  *error\_percent*.

The default values for *reject\_percent* and *error\_percent* are 100. If you omit only the *reject\_percent*, its default value is 100. If you omit only the *error\_percent*, its value is set to the *reject\_percent*. If the *reject\_percent* exceeds the *error\_percent*, a warning is issued and the *error\_percent* is reset to equal the *reject\_percent*. For example, the *error\_percent* in the following command line is reset to 100 because the *reject\_percent* has the default value of 100.

```
% ncelab top -pulse_e 80
```

In the following command line, the *error\_percent* is set to the *reject\_percent* of 50.

```
% ncelab top -pulse_r 50
```

Example:

This example shows how to use the *-pulse\_r* and *-pulse\_e* options to set global path pulse control. In this example, module path delay = 50 time units.

```
% ncelab -pulse_r 60 -pulse_e 90 hardrive
```

In this example, a module path delay of 50 time units has a reject limit of 30 time units (60% of 50 time units). The error limit is 45 time units (90% of 50 time units).

- Pulses smaller than 30 time units are rejected.
- At 30 through 44 time units, pulses are set to the error state and then mapped to the `x` state.
- At 45 time units and above, pulses are passed through.

Use the `-epulse_no_msg` option when you invoke the simulator to suppress the display of error messages for pulses smaller than `error_percent`.

```
% ncsim -epulse_no_msg top
```

Use the `-epulse_onevent` or `-epulse_ondetect` option when you invoke the simulator to specify the type of pulse filtering that you want to use. See “[Pulse Filtering Style](#)” on page 285 for more information.

## Pulse Control for Specific Modules and Module Paths

You can override global pulse control for module paths by declaring `PATHPULSE$` specparams in specify blocks. The `PATHPULSE$` specparam narrows the scope of module path pulse control to a specific module or to particular paths within modules.

Use the `-pathpulse` command-line option to enable the `PATHPULSE$` specparams.

**Note:** Standard Delay Format (SDF) annotation can provide new values for pulse limits of both module 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.

The syntax for the `PATHPULSE$` specparam is:

```
pulse_control_specparam
 ::= PATHPULSE$ = (reject_limit [,error_limit]);
 || PATHPULSE$module_path_source$module_path_destination =
      (reject_limit [,error_limit]);
```

If a module path source and a module path destination are specified, the pulse control is applied to the specific module path. If the source and destination are not included, the pulse control is applied to all paths declared within the module. If both path-specific `PATHPULSE$` specparams and a non-path-specific `PATHPULSE$` specparam are specified in the same module, the path-specific specparams take precedence.

The sources and destinations in `module_path_source` and `module_path_destination` can be scalar nets or vector nets, but they cannot be bit-selects or part-selects. The pulse handling characteristics you specify for paths beginning

in a vector and ending in a vector automatically apply to all module paths connecting the two vectors.

Values assigned to the `PATHPULSE$` specparam define the pulse handling windows in time units (not percentages, as in the `-pulse_r` and `-pulse_e` options). The first value represents the reject limit; the second value is the error limit. If you supply only one value, both limits are set to the same value.

Example:

The following example illustrates how to use `PATHPULSE$` specparams to set path pulse controls on specific paths. You must use the `-pathpulse` option when you elaborate this design.

```
specify
  (clk => q) = 12;
  (data => q) = 10;
  (clr, pre *> q) = 4;
specparam
  PATHPULSE$ = 3;
  PATHPULSE$clk$q = (2, 9);
  PATHPULSE$clr$q = 1;
endspecify
```

In this example:

- The second `PATHPULSE$` specparam sets a reject limit of 2 and an error limit of 9 for the path `(clk=>q)`.
- The third `PATHPULSE$` specparam sets reject and error values of 1 for the path `(clr*>q)`.

Note that a pulse control limit is specified for the first input signal `clr` in module path `(clr, pre => q)`, but that no pulse control limit is specified for `pre`, the second signal in the path. Pulse limits for this path are not affected by `PATHPULSE$clr$q`.

**Note:** In Verilog-XL, you must use the `+expand_specify_vectors` option to obtain this behavior. Without this option, all signals in module paths with multiple inputs or outputs have the same delays and pulse handling. That is, without the `+expand_specify_vectors` option, the third `PATHPULSE$` specparam sets reject and error values of 1 for both `clr=>q` and `pre=>q`.

- The first `PATHPULSE$` specparam sets reject and error values of 3 for the path `(data=>q)`.

## Pulse Filtering Style

The NC-Verilog simulator provides two methods of pulse filtering called *On-Event* and *On-Detect*.

On-Event filters pulses so that the transition to X occurs after the normally calculated delay for the originally scheduled transition. The transition from X occurs after the normally calculated delay for the new output state.

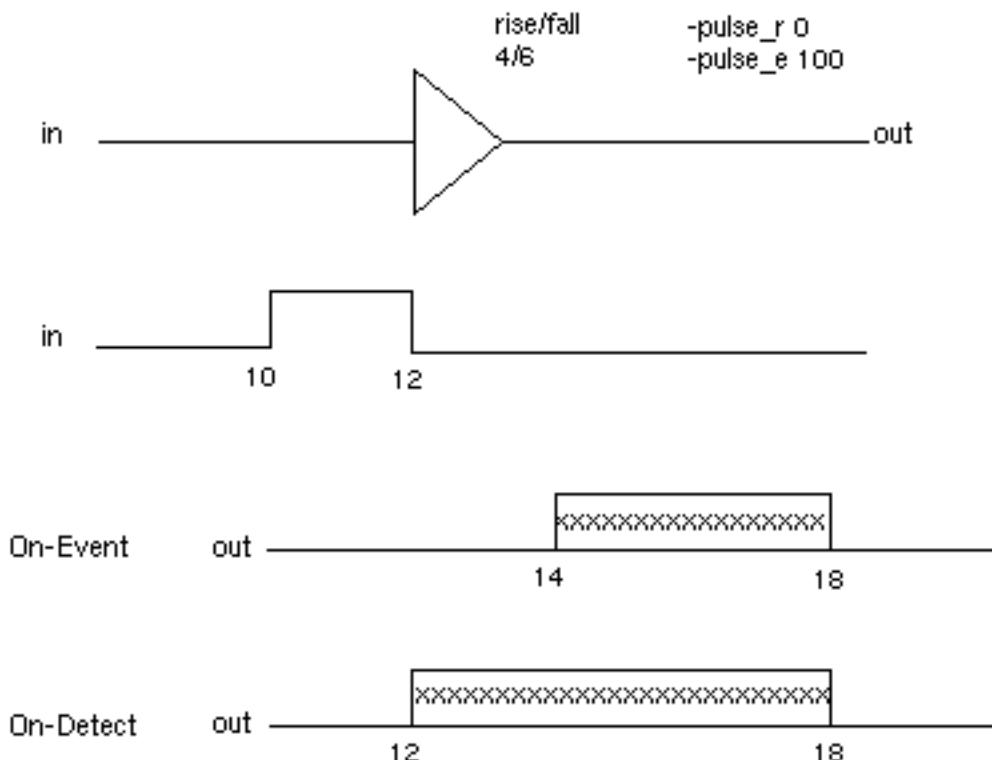
On-Detect filters pulses so that the transition to the X state occurs immediately upon the detection of the pulse error. The X state remains until the normally calculated delay for the new output state.

The On-Detect method allows more pessimism when filtering pulses to the X state, producing a longer X region. This method more closely reflects the output caused by nearly simultaneous inputs that result in scheduling output events at the same time.

This section includes an example that illustrates the difference between the On-Event and On-Detect methods of pulse filtering. It also discusses anomalies associated with schedule cancellation, which occurs when narrow pulses or nearly simultaneous transitions occur at model inputs.

### On-Event vs. On-Detect Pulse Filtering

The following figure uses a simple buffer with asymmetric rise/fall times and pulse limits equal to the delay to illustrate the difference between On-Event and On-Detect pulse filtering. An output waveform is shown for both On-Event and On-Detect.



With a rise delay of 4 and a fall delay of 6, the NC-Verilog simulator schedules the delay for times 14 and 18 based on the input transitions. If pulse limits are set equal to the delay, the simulator generates an error whenever a module path pulse is less than the module path delay. If you use the On-Event pulse filter, the X state region exists from time 14 to 18. If you use the On-Detect pulse filter, the X state region extends from the closing edge of the violating input transition (time 12) to the normally calculated delay for the new output state (time 18).

You can specify the pulse filtering style by using elaborator (*ncelab*) command-line options or by using keywords in a `specify` block. The command-line options specify the type of pulse filtering to use for all module path and interconnect delays. The `specify` block keywords let you specify the pulse filtering method to use for specific paths.

Command-line options override keywords in `specify` blocks. If you do not specify any keywords or command-line option, On-Event is the default.

The command-line options are:

- -epulse\_onevent (default)
- -epulse\_ondetect

### Example

```
% ncelab -epulse_ondetect top
```

The pulsestyle\_onevent and pulsestyle\_ondetect keywords can be used in a specify block to specify the pulse filtering style for particular paths. The syntax is:

```
pulsestyle_onevent path_output ...;  
pulsestyle_ondetect path_output ...;
```

The pulse filtering style keywords must be defined for an output prior to any path declarations for that output.

### Example 1:

In this example, no keywords are specified within the specify block. If no command-line option is specified, the default On-Event method is used.

```
specify  
  (a => out) = (2,3);  
  (b => out) = (3,4);  
endspecify;
```

### Example 2:

In this example, the pulsestyle\_ondetect keyword is used to apply the On-Detect pulse filtering method to outputs out and out\_b.

```
specify  
  pulsestyle_ondetect out;  
  (a => out)=(2,3);  
  (b => out)=(4,5);  
  pulsestyle_ondetect out_b;  
  (a => out_b)=(5,6);  
  (b => out_b)=(3,4);  
endspecify;
```

Because multiple output declarations are allowed for the same keyword, the following line could have been used in the above example:

```
pulsestyle_ondetect out, out_b;
```

### **Example 3:**

In the following example, the default On-Event is applied to output `out` prior to the (`a => out`) path statement. This is then followed by the `pulsestyle_ondetect` keyword, which applies the On-Detect method to `out`. This generates an error because an output path cannot have both methods applied to it.

```
specify
  (a => out)=(2,3);
  pulsestyle_ondetect out;
  (b => out)=(3,4);
endspecify;
```

### **Pulse Filtering and Cancelled Schedules**

A schedule is cancelled when a delay schedules a transition to occur before a previously scheduled transition. By default, the presence of canceled schedules is not indicated with an X state region. You can cause `specify` path outputs to use the X state to indicate the presence of cancelled schedules by using the `-epulse_neg` command-line option when you invoke the elaborator or by using keywords in the `specify` block.

The `-epulse_neg` option turns on the X state display for all `specify` paths. The default is `-epulse_noneg`.

#### **Example**

```
% ncelab -epulse_ondetect -epulse_neg top
```

Use the `showcancelled` and `noshowcancelled` keywords in a `specify` block to turn on the display for particular paths. The syntax is:

```
showcancelled path_output ...;
noshowcancelled path_output ...;
```

The keywords must be defined for an output prior to any path declarations for that output.

Command-line options override keywords in `specify` blocks. If you do not specify any keywords or command-line option, cancelled schedules are not indicated.

For example, in the following `specify` block, the `showcancelled` keyword turns on the X state display for cancelled schedules, and the `pulsestyle_ondetect` keyword specifies On-Detect style pulse filtering:

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

```
specify
    showcancelled out;
    pulsetestyle_ondetect out;
    (a => out)=(2,3);
    (b => out)=(4,5);

    showcancelled out_b;
    pulsetestyle_ondetect out_b;
    (a => out_b)=(5,6);
    (b => out_b)=(3,4);
endspecify;
```

Because multiple output declarations are allowed for the same keyword, the following specify block produces the same result as the example above:

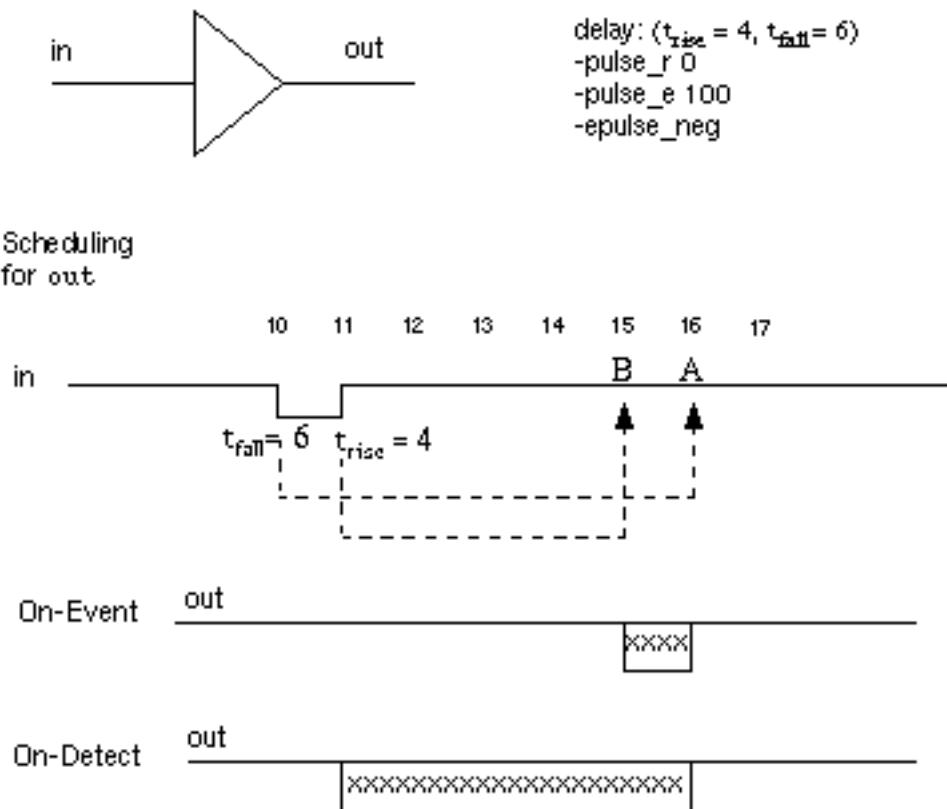
```
specify
    showcancelled out,out_b;
    pulsetestyle_ondetect out,out_b;
    (a => out)=(2,3);
    (b => out)=(4,5);
    (a => out_b)=(5,6);
    (b => out_b)=(3,4);
endspecify;
```

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

The following figure shows the X state region that occurs with a cancelled schedule for each method of pulse filtering. Neither On-Event nor On-Detect provides a “correct” answer. Select the method that you want to use based on your library characterization and best judgement.



The events in this figure occur as follows:

1. At time 10, a 1->0 transition on the input causes the NC-Verilog simulator to schedule event A at time 16 ( $10 + 6$ ).
2. At time 11, a 0->1 transition on the input causes the NC-Verilog simulator to schedule event B at time 15 ( $11 + 4$ ).
3. Because event B is scheduled to occur before event A, the schedule for A is cancelled. If you include the `-epulse_neg` option, an X state region is produced based on the pulse filtering method you use, as follows:
  - On-Event pulse filtering produces an X state region on `out` that begins at the time of the second schedule, B, and that ends at the time of the cancelled scheduled event, A, which is replaced with a schedule to the new logic state (in this case, 1).

## NC-Verilog Simulator Help

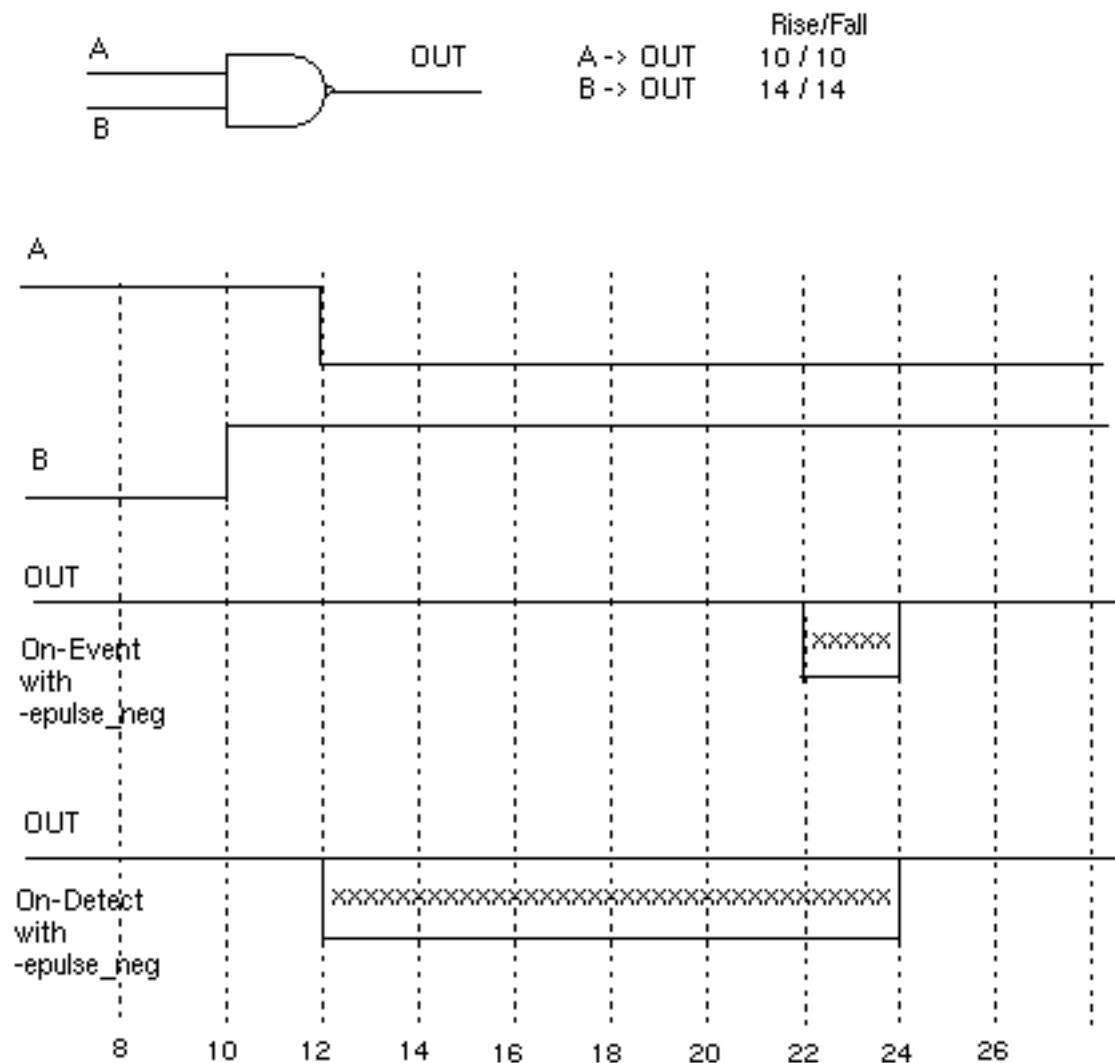
### Elaborating the Design with ncelab

- ❑ On-Detect pulse filtering produces an X state region on `out` that begins at the time the schedule was cancelled and ends at the time of the cancelled schedule A, which is replaced with a schedule to the new logic state (in this case, 1).

The following two examples show what happens when two inputs arrive at nearly the same time (closer together in time than the difference in delays) and cause a schedule cancellation. In the first example, the output events occur at different times; in the second example, the output events are scheduled at the same time.

#### **Example 1:** Nearly Simultaneous Switching Inputs (different event time)

The condition shown in this example is similar to the one described above in the buffer case, except that multiple signals are involved. The figure shows the waveform for a two-input NAND gate where input A is 1 and input B is 0.



At time 10, input B makes a 0->1 transition, which schedules the output to make the 1->0 transition at time 24. At time 12, input A transitions 1->0, scheduling the output to transition from 0->1 at time 22.

Because the second input (A) causes a schedule to be placed on the output prior to the one already scheduled from the B transition, the output takes on an X state to reflect the uncertainty of the output state between the two schedules.

If the input events cause the output to transition in the same direction, the X state is ignored because this is just a timing difference rather than an event that causes the output to create a momentary pulse.

**Example 2:** Nearly Simultaneous Switching Inputs (same event time)

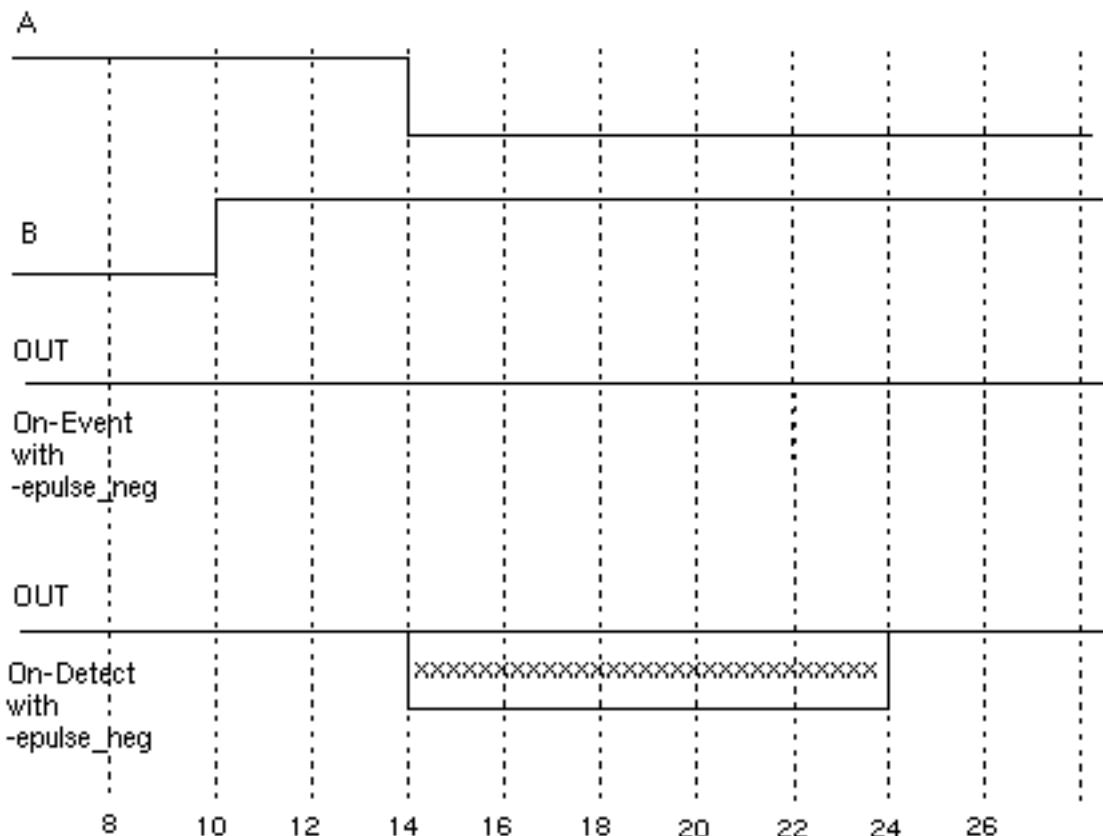
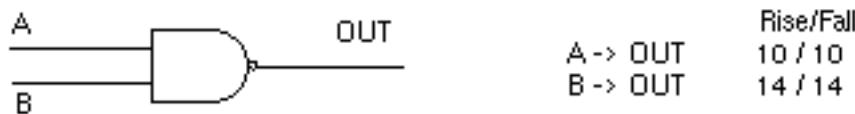
This example shows the result of nearly simultaneous input events causing output events to be scheduled at the same time. The transitions on inputs A and B both cause output events to be scheduled at time 24.

In this case, On-Event mode does not reflect that the event has occurred. This is because both output events occur at the same time, so no delta with an X can be shown. The

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

On-Detect mode provides a longer time period in which the uncertainty can be indicated with an X.



The following is a more complex example of cancelled schedules, using two inputs to `out`. This example also illustrates the effects of delay recalculation.

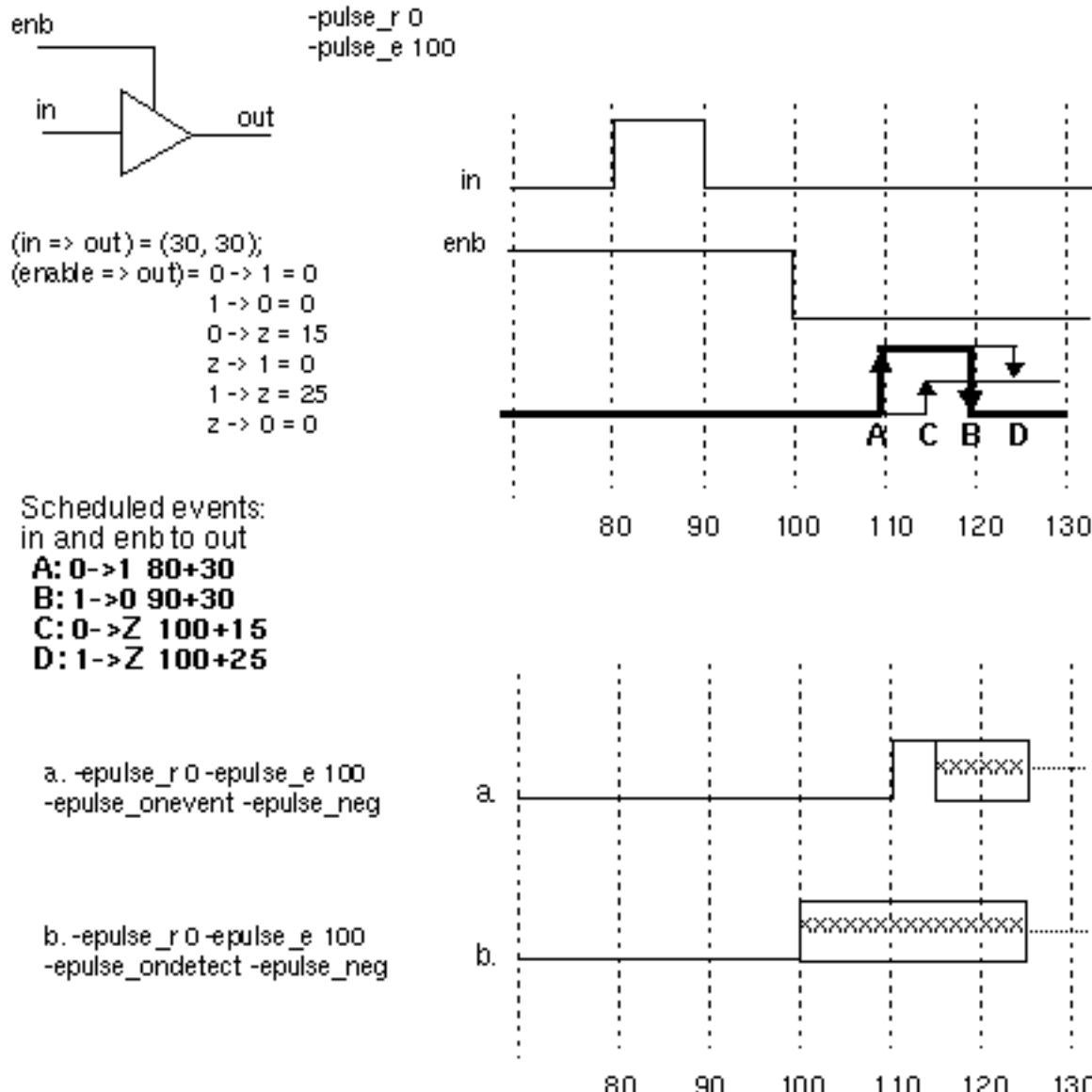
Delay recalculation takes place when a schedule is cancelled, causing a change in the state previous to the new state. The delay must be recalculated based on the new state from which the signal will be transitioning. The following example shows a delay being calculated for a 0

## NC-Verilog Simulator Help

### Elaborating the Design with ncelab

---

-> Z transition, but the schedule to 0 is cancelled, and so a 1 -> Z transition delay must be calculated.



The waveforms in the previous figures are explained as follows:

The transport delay algorithm recalculates the 0->z delay based on a 1->z transition. This changes the time of the transition to `z` on the output to time 125. However, if that change is done, then the 1->0 transition at time 120 no longer needs to be cancelled, producing a cancelled schedule dilemma.

## **NC-Verilog Simulator Help**

### Elaborating the Design with ncelab

---

Wave a is produced because the On-Event filter causes an `x` to appear on the output due to the `1->0` schedule at time 120 being cancelled. The `e` state extends from time 115 to 125.

Wave b is produced because the On-Detect filter extends the `e` state to the edge of the event that caused the pulse to occur, which is the transition on `enb` at time 100.

---

# Simulating Your Design with ncsim

---

This chapter contains the following sections:

- [Overview](#)
- [ncsim Command Syntax](#)
- [ncsim Command Options](#)
- [Example ncsim Command Lines](#)
- [hdl.var Variables](#)
  
- [Invoking the Simulator](#)
- [Starting a Simulation](#)
- [Saving, Restarting, Resetting, and Reinvoking a Simulation](#)
- [Updating Design Changes When You Invoke the Simulator](#)
- [Providing Interactive Commands from a File](#)
- [Exiting the Simulation](#)

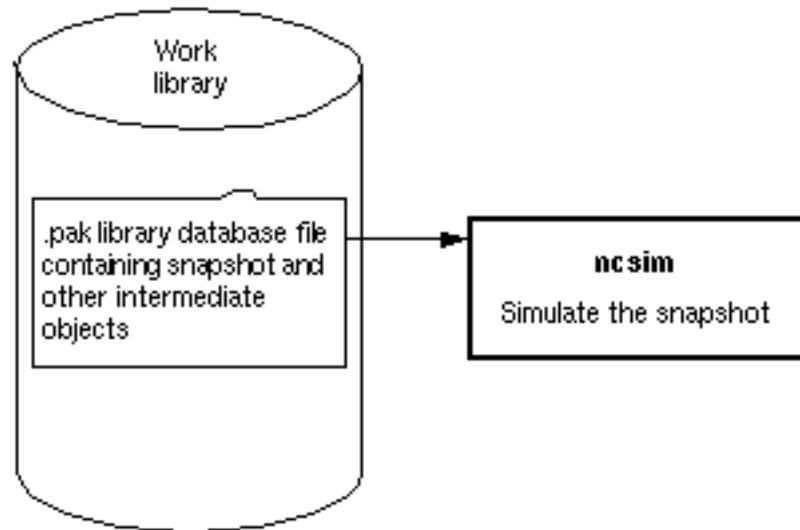
## Overview

After you have compiled and elaborated your design, you can invoke the simulator, *ncsim*. This tool simulates Verilog and VHDL using the compiled-code streams to execute the dynamic behavior of the design.

*ncsim* loads the snapshot as its primary input. It then loads other intermediate objects referenced by the snapshot. In the case of interactive debugging, HDL source files and script files also may be loaded. Other data files may be loaded as demanded by the model being simulated (via \$read\* tasks or Textio).

The outputs of simulation are controlled by the model or debugger. These outputs can include result files generated by the model, Simulation History Manager (SHM) databases, or Value Change Dump (VCD) files.

The following figure illustrates the *ncsim* process flow.



Invoke *ncsim* with options and a snapshot name specified in Lib.Cell:View notation. The options and the snapshot argument can occur in any order except that parameters to options must immediately follow the option they modify. Only one snapshot can be specified.

The syntax for invoking the simulator is:

```
% ncsim [options] [Lib.]Cell[:View]
```

- You must specify the cell.
- If a snapshot with the same name exists in more than one library, the easiest (and recommended) thing to do is to specify the library on the command line.

- If there are multiple views that contain snapshots, the easiest (and recommended) thing to do is to specify the view on the command line.

See [Chapter 8, “Elaborating the Design with ncelab,”](#) for information on how *ncelab* names snapshots.

### Rules for Resolving the Snapshot Reference

If you do not specify a library or a view, *ncsim* uses the following rules to resolve the snapshot reference on the command line (assuming that the command line is % ncsim top):

**1. Is the WORK variable set in the hdl.var file?**

```
YES => Does WORK.top exist?  
        YES => How many views of WORK.top have snapshots?  
                1 => Simulate this snapshot.  
                More than 1 => Error message  
                        (More than one snapshot matches "top")  
        NO => Go to Step 2.  
NO => Go to Step 2.
```

**2. Search all libraries in the cds.lib file.**

```
Does LIB*.top exist?  
        YES => How many views of LIB*.top have snapshots?  
                1 => Simulate this snapshot.  
                More than 1 => Error message  
                        (More than one snapshot matches "top".)  
        NO => Error message  
                        (Snapshot "top" does not exist in the libraries.)
```

## ncsim Command Syntax

The *ncsim* command-line options shown below are divided into three groups: General options, which apply to both languages, Verilog only options, which apply to Verilog portions of a design, and VHDL only options, which apply to VHDL portions of a design.

Options can be abbreviated to the shortest unique string, indicated here with capital letters.

```
ncsim [options] [Lib.]Cell[:View]
```

### General Options

```
[-APPEND Key]  
[-APPEND Log]  
[-Batch]  
[-Cdslib cdslib_pathname]  
[-ERRormax integer]  
[-EXIT]  
[-File arguments_filename]  
[-Gui]  
[-HDLvar hdlvar_pathname]  
[-HELP]  
[-Input script_file]  
[-Keyfile filename]  
[-LICqueue]  
[-LOGfile filename]  
[-Messages]  
[-NCError warning_code]  
[-NCFatal {warning_code | error_code}]  
[-NEverwarn]  
[-NOCopyright]  
[-NOKey]  
[-NOLICPromote]  
[-NOLICSuspend]  
[-NOLOg]  
[-NOSource]  
[-NOSTdout]  
[-NOWarn warning_code]  
[-OMICcheckinglevel checking_level]  
[-PPE]  
[-PROFILE]  
[-PROFThread]
```

# NC-Verilog Simulator Help

## Simulating Your Design with ncsim

---

[-REdmem]  
[-RUn]  
[-Status]  
[-Tcl]  
[-UNbuffered]  
[-UPdate]  
[-VCdextend]  
[-VErsion]

### Verilog Only Options

[-EPulse no msg]  
[-LOADVPI *shared\_library\_name*:*bootstrap\_function\_name*]  
[-NBasync]  
[-PLINOptwarn]  
[-PLINOWarn]  
[-Xlstyle units]

### VHDL Only Options

[-EXTassertmsg]  
[-LOADCfc [*CFC\_library*]:*bootstrap\_func\_name*[,*bootstrap\_func\_name*,...]]  
[-LOADFmi *FMI\_library*]  
[-LOADVHpi *shared\_library\_name*:*bootstrap\_function\_name*]  
[-NOCIfcheck]

## ncsim Command Options

This section describes the options that you can use with the `ncsim` command. Options can be entered in upper or lower case. Capital letters indicate the shortest possible abbreviation for an option.

The options listed below apply to both Verilog and VHDL unless noted otherwise.

### **-APPEND\_Key**

Append command input from multiple runs of *ncsim* to one key file. Use this option if you are going to run *ncsim* multiple times and you want all command input appended to one key file. If you do not use this option, the key file is overwritten each time you run *ncsim*.

By default, *ncsim* generates a key file called `ncsim.key` to capture command input. Use the [-keyfile](#) option to rename the key file.

Use `-nokey` if you don't want a key file.

### **-APPEND\_Log**

Append log information from multiple runs of *ncsim* to one log file. Use this option if you are going to run *ncsim* multiple times and you want all log information appended to one log file. If you do not use this option, the log file is overwritten each time you run *ncsim*.

If you use both `-append_log` and `-nolog` on the command line, `-nolog` overrides `-append_log`.

Because the log file is opened before variables in the `hdl.var` file are read, the `-append_log` option is ignored with a warning if you define it with the `NCSIMOPTS` variable in an `hdl.var` file.

### **-Batch**

Start the simulation or the processing of commands from `-input` options without waiting for command input. Use this option if you have included the `-tcl` option in your `NCSIMOPTS` variable in the `hdl.var` file because you invoke the simulator in interactive mode most of the time. Using `-batch` allows you to override `-tcl` and simulate in noninteractive mode. See ["Invoking the Simulator"](#) on page 322 for more information on invoking the simulator.

Example:

```
% ncsim -batch top
```

### **-Cdslib *cdslib\_pathname***

Use the specified `cds.lib` file. See “[The cds.lib File](#)” on page 110 for details on the `cds.lib` file.

All tools and utilities that require a `cds.lib` file use a default search mechanism to find the `cds.lib` file. See “[The setup.loc File](#)” on page 131 for information on this search mechanism. Use the `-cdslib` option to override the default search order and force the simulator to use the specified `cds.lib` file.

Example:

```
% ncsim -cdslib ~/design_lib/cds.lib top
```

*ncsim* reads the `cds.lib` file before it processes any variables defined in the `hdl.var` file. You cannot, therefore, include the `-cdslib` option with the `NCSIMOPTS` variable in an `hdl.var` file.

### **-EPulse\_no\_msg**

**(Verilog only)**

Suppress pulse control error messages. See “[Setting Pulse Controls](#)” on page 280 for more information.

### **-ERrormax *integer***

Abort after reaching the specified number of errors. By default, there is no limit on the number of error messages.

By using `-errormax`, you can limit the number of errors that are generated, fix those errors, and then rerun to check for other errors. This option is useful when you are running a large design that might contain numerous errors.

```
% ncsim -errormax 10 top
```

Errors caused by Tcl command files or by interactive Tcl commands do not count toward the `errormax` limit.

## **-EXIt**

Exit the simulation instead of entering interactive mode. Using this option guarantees that a noninteractive simulation will exit under conditions that would normally stop the simulation and return the *ncsim>* prompt.

## **-EXTassertmsg**

**(VHDL only)**

Print extended assert message information.

If you use this option, assert messages include additional information that tells you the location in your source code from which the function or procedure is being called.

The following example shows an assert message that was generated without the `-extassertmsg` option:

```
ASSERT/WARNING (time 902 NS) from package ieee.STD_LOGIC_ARITH, this builtin  
function called from function @ieee.std_logic_signed: "="  
Built-in relational argument contains a ('U', 'X', 'W', 'Z', '-') in an operand.
```

The following example shows the same assert message generated with the `-extassertmsg` option:

```
ASSERT/WARNING (time 902 NS + 1) from package ieee.STD_LOGIC_ARITH, this builtin  
function called from function @ieee.std_logic_signed: "=", process  
:TOP:TDSP_CORE_INST:ALU_32_INST:GENERATE_OVERFLOW (architecture  
WORKLIB.alu_32:rtl)  
Built-in relational argument contains a ('U', 'X', 'W', 'Z', '-') in an operand.
```

## **-File arguments\_filename**

Use the command-line arguments contained in the specified arguments file.

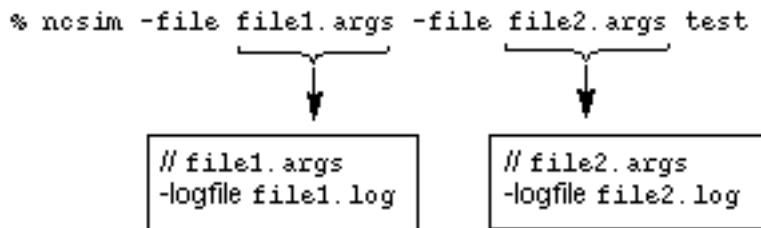
You can store frequently used or lengthy command lines by putting command-line arguments (command options and the snapshot name) in a text file. When you invoke *ncsim* with the `-file` option, the arguments in the arguments file are incorporated with your command as if they had been entered on the command line.

The arguments file can contain command options, including other `-file` options, and the snapshot name. The individual arguments within the arguments file must be separated by white space or comments.

**Example:**

```
% ncsim -file ncsim.args top
```

If you use the same command-line option with different arguments in different argument files, a fatal message is generated. For example, you cannot do the following:



**-Gui**

Invoke the simulator with the SimVision analysis environment.

The following command invokes *ncsim* with the SimVision analysis environment and automatically stops the simulation at time 0:

```
% ncsim -gui top
```

The following command invokes *ncsim* with the analysis environment and automatically starts the simulation or the processing of commands from `-input` options without prompting you for command input:

```
% ncsim -gui -run top
```

See “[Invoking the Simulator](#)” on page 322 for more information on invoking the simulator.

**Note:** On Windows, you cannot invoke the simulator with the SimVision analysis environment by including the `-gui` option in the definition of the `NCSIMOPTS` variable in the `hdl.var` file.

**-HDLvar *hdlvar\_pathname***

Use the specified `hdl.var` file. See “[The hdl.var File](#)” on page 118 for details on the `hdl.var` file.

All tools and utilities that require an `hdl.var` file use a default search mechanism to find the `hdl.var` file. See “[The setup.loc File](#)” on page 131 for information on this search mechanism. Use the `-hdlvar` option to override the default search order and force the simulator to use the specified `hdl.var` file.

Example:

```
% ncsim -hdlvar ~/hdl.var top
```

You cannot include this option in an `hdl.var` file with the `NCSIMOPTS` variable.

### **-HElp**

Display a list of the `ncsim` command options with a brief description of each option.

```
% ncsim -help
```

This option is ignored if you include it with the `NCSIMOPTS` variable in an `hdl.var` file.

### **-Input *script\_file* (or: -Input @*tcl\_command*)**

Execute the Tcl commands in the specified script file (or execute the specified Tcl command) at the beginning of the simulation session.

You can also execute a script file of Tcl commands by using the `input` command or the `source` command. See “[input](#)” on page 560 for details on the `input` command. See “[source](#)” on page 616 for details on the `source` command.

See “[Providing Interactive Commands from a File](#)” on page 335 for more information on different ways to execute commands in a script file.

### **-Keyfile *filename***

Use the specified name for the key file instead of the default name `ncsim.key`.

Example:

```
% ncsim -keyfile top.key top
```

Key files are useful when you want to reproduce a simulation session. To reproduce an interactive session, specify the name of the key file with the `-input` option. The commands in the key file are executed at the beginning of the simulation session. When `ncsim` has processed all commands in the file, or if you interrupt processing with CTRL/C, input reverts back to the terminal.

**Note:** A key file contains all interactive commands that you have issued, including misspelled commands and the `exit` command. You may have to edit the key file before you can use it as an input file.

Use `-nokey` if you don't want a key file.

If you use both `-keyfile` and `-nokey` on the command line, `-keyfile` overrides `-nokey`.

Use [append key](#) if you are going to run *ncsim* multiple times and you want all command input appended to one key file. If you do not use this option, the key file is overwritten each time you run *ncsim*.

### **-Llcqueue**

Queue the request for a simulator (*ncsim*) license if one is not currently available and run the simulation when a license becomes available.

### **-LOADCfc [CFC\_library]:bootstrap\_func\_name[,bootstrap\_func\_name,...]**

**(VHDL only)**

Dynamically load the specified CFC (C Function Call) library.

Use the `-loadcfc` option when you want to dynamically load a dynamic library containing a CFC application or multiple applications. You can use multiple `-loadcfc` options on the command line. Any number of shared objects containing CFC applications can be loaded using this option.

The arguments to the `-loadcfc` option are:

- *CFC\_library*—A simple name or a full-path-name (with or without file extensions) of the dynamic link library containing the CFC application. This argument is optional.
- *bootstrap\_func\_name*—An externally visible function in the dynamically linked CFC library which returns a pointer to the `cfcTable`. You must specify at least one bootstrap function. There can be multiple bootstrap functions in a single CFC library.

See [CFC Functions in the NC-VHDL Simulator C Interface User Guide](#) for information on building dynamic libraries and appending their full-path location to the path environment variable.

See [Loading CFC Applications from the Command Line](#) in the [NC-VHDL Simulator C Interface User Guide](#) for detailed information on how to use the `-loadcfc` command-line option.

**-LOADFmi [*FMI\_library*]:*bootstrap\_func\_name*[,*bootstrap\_func\_name*,...]**  
**(VHDL only)**

Dynamically load the specified FMI (Foreign Model Import) library.

Use the `-loadfmi` option when you want to dynamically load a dynamic library containing user FMI models. You can use multiple `-loadfmi` options on the command line. Any number of libraries that contain foreign models can be loaded into the simulator using this option.

The arguments to the `-loadfmi` option are:

- *FMI\_library*—A simple name or a full-path-name (with or without file extensions) of the dynamically linked FMI library. This argument is optional.
- *bootstrap\_func\_name*—An externally visible function in the dynamically linked FMI library which returns a pointer to the `fmiLibraryTable` array. You must specify at least one bootstrap function. There can be multiple bootstrap functions in a single FMI library.

See [Foreign Model Integration](#) in the *NC-VHDL Simulator C Interface User Guide* for information on building dynamic FMI libraries and appending their full-path location to the path environment variable.

See [Loading FMI Libraries from the Command Line](#) in the *NC-VHDL Simulator C Interface User Guide* for detailed information on how to use the `-loadfmi` command-line option.

**-LOADVHpi *shared\_library\_name*:*bootstrap\_function\_name***

**(VHDL only)**

Dynamically load the specified VHPI application.

Use the `-loadvhpi` option when you have already compiled a VHPI application into a shared dynamic library.

The argument to the `-loadvhpi` option is the name or full path of the shared library that contains the VHPI application, followed by the name of the bootstrap function. This function is part of the VHPI application, and is defined in the shared library.

The file extension of the shared library is optional and depends on the OS you are using. For example, if you are running on the Solaris platform, the library is called `library_name.so`.

For more information on the `-loadvhpi` option, see [Creating a Shared Dynamic Library](#) in the *VHPI User Guide*.

**-LOADVPI *shared\_library\_name:bootstrap\_function\_name***

**(Verilog only)**

Dynamically load the specified VPI application.

If a VPI application has already been compiled into a dynamic shared library, you can use `-loadvpi` to load the library and to register the system tasks defined in the application at run time.

**Note:** You cannot dynamically load a PLI1.0 application when you invoke the simulator because PLI1.0 applications require a system task or function. To dynamically load a PLI1.0 application, use the `-loadpli1` option when you elaborate the design (`ncelab -loadpli1`).

The argument to the `-loadvpi` option is the name or full path of the shared library that contains the VPI application followed by the name of the function that contains the calls to `vpi_register_systf()`, which register the new system tasks for the VPI application. This function, called the *bootstrap* function, is part of the VPI application, and is defined in the shared library.

The file extension of the shared library is optional. The elaborator appends a suffix that is consistent with the OS that you are running. For example, if you are running on the SUN4v platform and enter the following command, the elaborator searches for a library called `SSI.so`.

```
% ncsim -loadvpi SSI:ssi_boot top
```

See “Loading VPI Applications from the Command Line” in the *VPI User Guide and Reference* for more information.

**-LOGfile *filename***

Use the specified name for the log file instead of the default name `ncsim.log`.

Example:

```
% ncsim -logfile counter.log counter
```

Use `-nolog` if you don't want a log file. If you use both `-logfile` and `-nolog` on the command line, `-logfile` overrides `-nolog`.

Use [append log](#) if you are going to run `ncsim` multiple times and you want all log information appended to one log file. If you do not use this option, the log file is overwritten each time you run `ncsim`.

Because the log file is opened before variables in the `hdl.var` file are read, the `-logfile` option is ignored with a warning if you define it with the `NCSIMOPTS` variable in an `hdl.var` file.

## **-Messages**

Print informative messages during execution.

Example:

```
% ncsim -messages top
```

By default, simulator messages are printed to a log file called `ncsim.log`. Use `-logfile` to rename the log file. Use `-nolog` if you don't want a log file.

Messages are also printed to the screen by default. Use `-nostdout` if you want to suppress printing to the screen.

## **-NBasync**

### **(Verilog only)**

Places nonblocking assign events after read/write synchronization callbacks (registered by `tf_synchronize` and `tf_isynchronize`) at the end of the event queue.

Use the `+nbasync` option if you are running NC-Verilog in single-step mode with the `ncverilog` command.

By default, when you create a PLI 1.0 application that uses `misctf` routines and associate it with a user-defined system task or function, the `tf_synchronize()` and `tf_isynchronize()` routines schedule a callback in read-write mode to the `misctf` routine. The callback occurs after all events, including nonblocking assignments, in the current simulation time step have been processed.

Without the `-nbasync` option, `tf_synchronize()` and `tf_isynchronize()` fire at the same time as `readwrite_sync`. Using the `-nbasync` option allows the synchronization routines to be triggered at the same time as the Non-Blocking-Assignments are scheduled and not at the normal `readwrite_sync` time. This may change possible race condition output of the simulation.

### **-NCError *warning\_code***

Increase the severity level of the specified warning message from warning to error. The *warning\_code* argument is the message code (mnemonic) that appears in the warning message following the severity code.

Example:

```
% ncsim -ncerror ABCDEF worklib.top:module
```

You can include multiple `-ncerror` options on the command line.

Using this option can change the behavior of the tool because functions that return errors instead of warnings may behave differently. Warnings that are changed to errors are counted in the error count limit that you specify with the `-errormax` option.

### **-NCFatal {*warning\_code* | *error\_code*}**

Increase the severity level of the specified warning message or error message from warning or error to fatal. The *warning\_code* or *error\_code* argument is the message code (mnemonic) that appears in the message following the severity code.

Example:

```
% ncsim -ncerror LMNOPQ worklib.top:module
```

You can include multiple `-ncfatal` options on the command line.

### **-NEverwarn**

Disable printing of all warning messages.

Example:

```
% ncsim -neverwarn top
```

To turn off one or more specific warning messages, use `-nowarn`.

### **-NOClfcheck**

#### **(VHDL only)**

Disable constraint checking in VHDL Design Access (VDA) functions for increased performance.

The VDA library is a C interface library that provides a mechanism for the interaction between other C interface libraries and the objects, scopes, and data types in the VHDL model. The VDA also contains routines for examining and manipulating the values of VHDL objects, as well as for setting up callbacks on signal events and simulation times. See Chapter 6, “The VHDL Design Access Library,” in the [Cadence NC-VHDL Simulator C Interface User Guide](#) for details on VDA functions.

### **-NOCCopyright**

SUPPRESS printing of the copyright banner.

Because the copyright banner is displayed before any variables in the `hdl.var` file are processed, this option is ignored if you include it with the `NCSIMOPTS` variable in an `hdl.var` file.

### **-NOKey**

DO NOT generate a key file. By default, `ncsim` generates a key file called `ncsim.key`.

The `-nokey` option is overridden by the `keyfile` option.

### **-NOLICPromote**

Turn off automatic license promotion.

The simulator detects which languages are in a snapshot. If only one language is present, it checks out the relevant license. If that license is not available, the simulator attempts to use an NC-Sim license (or an NC-Sim Desktop license if you are running the Desktop simulator). If you use `-nolicpromote`, the simulator does not try to check out the mixed-language license.

If both Verilog and VHDL are present in the snapshot, the simulator attempts to use an NC-Sim license (or an NC-Sim Desktop license if you are running the Desktop simulator). If that license is not available, the simulator attempts to use one NC-Verilog (or Verilog Desktop) license and one NC-VHDL (or VHDL Desktop) license. If you use `-nolicpromote`, the simulator does not try to check out these licenses.

### **-NOLICSuspend**

DO NOT release the simulator license(s) when suspending a job running under Platform Computing’s Load Sharing Facility® (LSF) software. If you are running NC-Verilog in

single-step invocation mode with the `ncverilog` command, use the `+nolicsuspend` option.

**Note:** License suspension is not supported on Windows. Using the `-nolicsuspend` option on Windows generates a message saying that this is the default.

Simulation jobs being managed by LSF can be suspended by LSF if the software determines that a higher priority job must be run. You can also suspend a running job by typing Control-Z. When a job suspension is requested from LSF or by a Control-Z, the simulator, by default, releases any licenses before suspending the job so that the higher-priority job will not be denied a license because it is being consumed by the suspended low-priority job.

**Note:** In LSF, the default signal for job suspension is `SIGSUSP`. This signal cannot be blocked or caught by the simulator, so the simulator cannot respond and release licenses before a job is suspended. You must configure LSF so that the notification of request for suspension is a `SIGTSTP` signal, the same signal that is used for Control-Z from the keyboard. You can do this by configuring the queue on which the simulation jobs are run. Add the following entry to the queue definition:

```
JOB_CONTROLS = SUSPEND[SIGTSTP]
```

When the suspended job is resumed by LSF (or by typing `f_g` in a shell window), all licenses are checked out again, and license queuing is enabled so that a resumed job will not exit if no licenses are available. Forcing queuing in this way causes the job to resume simulating only after all licenses become available.

The default mode (`-licssuspend`) only affects the simulator when running jobs controlled by LSF, and you will probably never need to turn off the default behavior. The `-nolicsuspend` option is provided if you need to change the default behavior for some reason.

### **-NOLOG**

Do not generate a log file. By default, `ncsim` generates a log file called `ncsim.log`.

The `-nolog` option is overridden by the [`-logfile`](#) option.

### **-NOSOURCE**

Do not check source file timestamps when using the `-update` option. See [“Updating Design Changes When You Invoke the Simulator”](#) on page 332 for more information.

### **-NOSTdout**

Suppress the printing of most output to the screen.

If you are using *ncsim* in interactive mode, the `-nostdout` option turns off output generated from the model via, for example, Verilog `$monitor` or `$display` commands or VHDL assert messages or textio writes to standard output. The *ncsim>* prompt, results of simulator commands, and a few other messages are still printed to the screen when you specify `-nostdout`.

The `-nostdout` option does not change what is written to the log file.

### **-NOWarn *warning\_code***

Disable printing of the warning with the specified code. The *warning\_code* argument is the message code (mnemonic) that appears in the warning message following the error severity code.

Example:

```
% ncsim -nowarn HVAPKF top
```

You can include multiple `-nowarn` options on the command line.

### **-Omicheckinglevel *checking\_level***

Specify OMI checking level. The *checking\_level* argument can be:

- `max`—maximum checking level. Use this level for early integration testing and to debug problems.
- `std`—standard checking level. This is the default.
- `min`—minimum checking level. Select this level to achieve higher performance after problems have been debugged.

See “[The Open Model Interface \(OMI\)](#)” on page 952 for details on OMI.

### **-PLINOOptwarn**

#### **(Verilog only)**

Display only one warning message the first time that a PLI read, write, or connectivity access violation is detected.

By default, the simulator displays all warning and error messages generated when an error is detected due to a PLI access violation. Use this option to suppress the display of these access violation messages. If you use this option, a warning message is displayed once, when the first access violation is detected. The message is displayed again if an access violation is detected after a reset or a restart has been executed.

Example:

```
% ncsim -plinooptwarn top
```

### **-PLINOWarn**

#### **(Verilog only)**

Suppress the display of PLI warning and error messages. These messages are displayed by default.

Example:

```
% ncsim -plinowarn top
```

### **-PPe**

Invoke the Post Processing Environment (PPE).

The PPE lets you analyze simulation results stored in an SHM database and debug your design without using a simulator license. The PPE gives you access to all of the SimVision analysis environment tools that are available in interactive mode. The features of each tool are virtually identical to those in interactive mode.

See “[The SimVision Post Processing Environment](#)” in the *SimVision Analysis Environment User Guide* for details on using the PPE.

## **-PROFIle**

Generate a run-time profile of the design.

The `-profile` option generates a file called `ncprof.out` in the current working directory. This file contains simulation runtime information that is useful for finding performance bottlenecks and for tuning a design description for better simulation performance.

Include the `-profthread` option if you have threaded C applications.

See “[Using the Profiler to Identify and Eliminate Simulation Bottlenecks](#)” on page 697 for more information on the profiler.

## **-PROFThread**

Enable the profiling of threaded processes.

Use this option with the `-profile` option if you have C applications that are threaded.

## **-REdmem**

Do not load the intermediate objects generated by the *ncvlog* or *ncvhdl* compiler when the simulator is invoked.

By default, *ncsim* loads these intermediate objects to enable access to source code for debugging the design and for providing PLI code with access to the design data. However, *ncsim* does not require access to this data to simulate the design. If you use this option, the intermediate objects are loaded from the library database only when information about the design data is requested. Depending on the amount of data in the intermediate objects, this can result in significantly lower memory consumption.

The performance impact of using `-redmem` is minimal in most cases. For a design that includes a significant amount of PLI code, the impact may be more severe.

Do not recompile while the simulation is running with `-redmem`. Recompilation causes changes to the data in the library database, and the simulation may fail because the intermediate objects being requested no longer exists.

## **-RUn**

Execute the simulation after initialization without waiting for user input.

Use this option if you want to invoke *ncsim* in noninteractive mode. This option is optional if you are invoking the simulator without the SimVision analysis environment. The following two command lines are equivalent:

```
% ncsim -run top  
% ncsim top
```

The *-run* option must be used if you want to invoke the simulator in noninteractive mode with the analysis environment.

```
% ncsim -gui -run top
```

If you do not use the *-run* option, the simulator is invoked with the analysis environment and stops at time 0.

If you want to invoke *ncsim* in noninteractive mode, but do not want to exit the simulator when the simulation ends, include the *-tcl* option. The *-tcl* option tells the simulator to enter interactive mode, while the *-run* option tells it to start the simulation without waiting for command input.

Example:

```
% ncsim -run -tcl top
```

See “[Invoking the Simulator](#)” on page 322 for more information on invoking the simulator.

## **-Status**

Print statistics on memory and CPU usage after simulation.

The following example shows the output of the *-status* option:

```
ncsim: Memory Usage - 8.3M program + 1.7M data = 10.0M total  
ncsim: CPU Usage - 1.0s system + 0.7s user = 1.7s total (1.3s, 100.0% cpu)
```

## **-Tcl**

Invoke the simulator in interactive mode. This option invokes the simulator and stops at time 0 so that you can start entering interactive commands.

Example:

```
% ncsim -tcl top
```

If you are invoking *ncsim* with the SimVision analysis environment, *ncsim* automatically stops at the beginning of the simulation. The *-tcl* option is not required.

The following two commands are equivalent.

```
% ncsim -gui top  
% ncsim -gui -tcl top
```

See “[Invoking the Simulator](#)” on page 322 for more information on invoking the simulator.

### **-UNbuffered**

Do not buffer output.

This option forces output to bypass the file I/O buffer. This includes commands and statements that display information and that write information to log files key files, and so on.

The `-unbuffered` option is useful when you need simulation data to be displayed as soon as it is generated. Otherwise, the simulator waits for the buffer to fill and then outputs the data.

The `-unbuffered` option imposes a significant performance penalty. Use `-unbuffered` only when you need output information to bypass the buffer.

Example:

```
% ncsim -unbuffered top
```

### **-UPdate**

Recompile any out-of-date design units if needed.

When you change design units in the hierarchy, you must recompile them and re-elaborate the design hierarchy.

Use the `-update` option to automatically recompile and re-elaborate all out-of-date design units. This option calls *ncupdate* to:

- Recompile any changed design units.
- Recompile SDF source files if they have changed.
- Re-elaborate the design.

**Note:** See “[NCUPDATEOPTS](#)” on page 123 for information on specifying the path to an alternate elaborator.

- Generate a new snapshot.

The `-update` option then invokes the simulator and loads the new snapshot.

See “[ncupdate](#)” on page 928 for details on *ncupdate*.

Use the `-nosource` option with `-update` if you recompile selected parts of your design and then want to automatically re-elaborate and load the new snapshot into the simulator.

See “[Updating Design Changes When You Invoke the Simulator](#)” on page 332 for examples of using the `-update` option.

### **-VCdextend**

Left-extend all vectors in VCD files.

By default, VCD eliminates redundant bit values that result from left-extending values to fill a particular vector size so that vector values appear in the shortest possible form. Some applications require that vectors be left-extended. Use this option to print the whole vector when any bit in the vector changes value. The rules for left-extending vector values are as follows:

<b>When the value is:</b>	<b>VCD left-extends with:</b>	<b>Example (binary value extended to fill a 4-bit register)</b>
1	0	10 extended to 0010
0	0	01 extended to 0001
Z	Z	ZX0 extended to ZZX0
X	X	X10 extended to XX10

### **-VErsion**

Print the version of *ncsim* and exit.

This option is ignored if you include it in an `hdl.var` file with the `NCSIMOPTS` variable.

### **-Xlstyle\_units**

#### **(Verilog only)**

Print time values using the same formatting rules that Verilog-XL uses. XL follows any `$timeformat` that is in effect, and, if no `$timeformat` is in effect, formats values to the smallest ``timescale` precision.

This option applies primarily to simulator messages that include time values, such as timing violation, charge decay, pulse width, \$finish/\$stop, and Tcl messages. Using this option can make it easier to compare simulation results from the two simulators.

This option affects the formatting of time values only. It does not affect the format of the messages.

You can also enable XL-style time formatting by setting the predefined `display_unit` variable. See [“Setting Variables”](#) on page 481 for information on predefined variables.

## Example ncsim Command Lines

The following command invokes the simulator in noninteractive mode. This command automatically starts the simulation or the processing of commands from `-input` options without prompting you for command input.

```
% ncsim top
```

The following command invokes the simulator in noninteractive mode with the SimVision analysis environment. The `-run` option is required. This command automatically starts the simulation or the processing of commands from `-input` options without prompting you for command input.

```
% ncsim -gui -run top
```

The following command invokes *ncsim* in interactive mode. The simulation stops at time 0.

```
% ncsim -tcl top
```

The following command invokes *ncsim* in interactive mode with the SimVision analysis environment. The simulation stops at time 0.

```
% ncsim -gui top
```

In the following command, the `-logfile` option renames the log file from the default (`ncsim.log`) to `top.log`.

```
% ncsim -messages -run -logfile top.log top
```

In the following example, `-errormax 10` tells the simulator to abort after 10 errors.

```
% ncsim -errormax 10 top
```

The following example uses the `-file` option to include a file called `top.vc`, which includes a set of command line options, such as `-messages`, `-nocopyright`, `-logfile`, and `-errormax`.

```
% ncsim -file top.vc top
```

In the following example, the `-input` option sources the file `top.inp` at initialization. This file contains a sequence of simulator (Tcl) commands.

```
% ncsim -input top.inp top
```

In the following example, the `-keyfile` option specifies that the name of the key file is `top.key` instead of the default `ncsim.key`. You could then use this file to reproduce an interactive session by using the file name `top.key` as the argument to the `-input` option.

```
% ncsim -tcl -keyfile top.key top
```

## hdl.var Variables

The following variables are used by *ncsim*:

- **NCSIMOPTS**

Sets simulator command-line options. A snapshot name can also be included.

**Example:**

```
DEFINE NCSIMOPTS -messages
```

- **WORK**

Specifies the default library in which to look for the snapshot. If the snapshot is not found in this library, the rest of the libraries in the `cds.lib` file are searched.

See “[The hdl.var File](#)” on page 118 for more information on the `hdl.var` file.

## Invoking the Simulator

You can invoke the simulator (*ncsim*) in two modes:

- Noninteractive mode

Automatically starts the simulation or the processing of commands from `-input` options without prompting you for command input. See “[Invoking the Simulator in Noninteractive Mode](#)” on page 323.

- Interactive mode

Stops the simulation at time 0 and returns the *ncsim>* prompt. See “[Invoking the Simulator in Interactive Mode](#)” on page 324.

In either mode, you can invoke the simulator with or without the SimVision analysis environment.

The syntax for invoking the simulator is:

```
% ncsim [options] snapshot
```

The options and the snapshot argument can occur in any order except that parameters to options must immediately follow the option they modify.

The *snapshot* argument is a Lib.Cell:View specification. You must specify the cell.

- If a snapshot with the same name exists in more than one library, the easiest (and recommended) thing to do is to specify the library on the command line.
- If there are multiple views that contain snapshots, the easiest (and recommended) thing to do is to specify the view on the command line.

If you do not specify a library or a view, *ncsim* uses a set of rules to resolve the snapshot reference on the command line. See “[Rules for Resolving the Snapshot Reference](#)” on page 298.

Only one snapshot can be specified.

See the “[Overview](#)” on page 181 for information on how *ncelab* names snapshots.

## Invoking the Simulator in Noninteractive Mode

Invoking *ncsim* in noninteractive mode automatically starts the simulation or processing of commands from *-input* options without prompting you for input.

- To invoke *ncsim* in noninteractive mode without the SimVision analysis environment, type:

```
% ncsim [-run] [other_options] snapshot
```

Examples:

```
% ncsim top  
% ncsim -run top
```

The *-run* option is not required.

- To invoke *ncsim* in noninteractive mode with the SimVision analysis environment, type:

```
% ncsim -gui -run [other_options] snapshot
```

Example:

```
% ncsim -gui -run top
```

The *-run* option is required.

**Note:** On Windows, you cannot invoke the simulator with the SimVision analysis environment by including the *-gui* option in the definition of the NCSIMOPTS variable in the *hdl.var* file.

If you have included the *-tcl* option in your NCSIMOPTS variable in the *hdl.var* file because you invoke the simulator in interactive mode most of the time, use *-batch* to override *-tcl* and simulate in noninteractive mode.

Example:

```
% ncsim -batch top
```

If you want to run in noninteractive mode, but do not want the simulator to exit at the end of the simulation, use both the *-run* and *-tcl* options. The following combination of options runs *ncsim* in noninteractive mode, but returns the *ncsim>* prompt instead of exiting:

```
% ncsim -run -tcl snapshot
```

Include the *-exit* option on the command line if you want the simulator to exit under conditions that would normally stop the simulation and put it in interactive mode.

The following examples illustrate the various options for invoking the simulator in noninteractive mode. Other `ncsim` command options are not shown.

% <code>ncsim [-run] top</code>	Invokes <i>ncsim</i> and runs the simulation.
% <code>ncsim -gui -run top</code>	Invokes <i>ncsim</i> with the SimVision analysis environment and runs the simulation.
% <code>ncsim -run -tcl top</code>	Invokes <i>ncsim</i> and runs the simulation.
	When the simulation is completed or interrupted, returns the <i>ncsim&gt;</i> prompt instead of exiting.
% <code>ncsim -batch top</code>	Use <code>-batch</code> if you want to simulate in noninteractive mode, but have the <code>-tcl</code> option included with the <code>NCSIMOPTS</code> variable in the <code>hdl.var</code> file.

## Invoking the Simulator in Interactive Mode

You can interact with your design throughout a simulation by instructing *ncsim* to stop at the beginning of the simulation so that you can enter interactive mode at simulation time 0.

If you are using the command-line interface, use the `-tcl` option when you invoke the simulator.

```
% ncsim -tcl [other_options] snapshot
```

**Example:**

```
% ncsim -tcl top
```

If you are using the SimVision analysis environment, *ncsim* automatically stops at the beginning of the simulation.

```
% ncsim -gui [-tcl] [other_options] snapshot
```

**Examples:**

```
% ncsim -gui top  
% ncsim -gui -tcl top
```

**Note:** The `-tcl` option is not required.

## Starting a Simulation

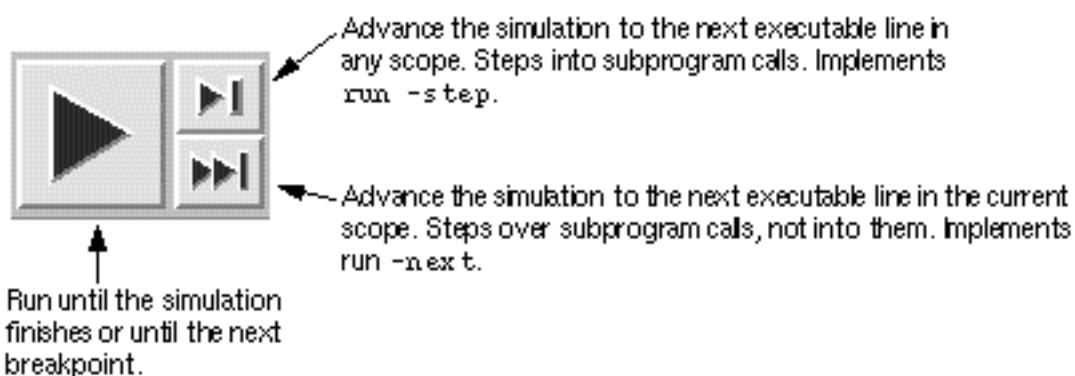
To start or resume a simulation:

- If you are using the Tcl command-line interface, use the `run` command. This command has several options that let you control when the simulation is to stop:
  - ❑ `-delta`—Run to the beginning of the next delta cycle or to a specified delta cycle.
  - ❑ `-next`—Run one behavioral statement, stepping over any subprogram calls.
  - ❑ `-return`—Run until the current subprogram (task, function, procedure) returns.
  - ❑ `-step`—Run one behavioral statement, stepping into subprogram calls.
  - ❑ `-timepoint`—Run for a specified number of time units.
  - ❑ `-phase`—Run to the beginning of the next phase of the simulation cycle. The two phases of a simulation cycle are signal evaluation and process execution.
  - ❑ `-process`—Run until the beginning of the next scheduled process or to the beginning of the next delta cycle, whichever comes first. In VHDL, a process is a process statement. In Verilog, it is an `always` block, `initial` block, or one of several kinds of anonymous behavior that can be scheduled to run.

See “[run](#)” on page 598 for details on the `run` command and for examples.

- If you are using the SimVision analysis environment, use the commands on the *Control* menu on the SimControl window. To simulate for a specified number of time units, set a time breakpoint before starting the simulation. See “[Setting a Time Breakpoint](#)” on page 430.

You can also use the following buttons on the SimControl tool bar:



## **NC-Verilog Simulator Help**

### Simulating Your Design with ncsim

---

See “[Starting a Simulation](#)” in the *SimVision Analysis Environment User Guide* for an example of starting and resuming a simulation from the SimControl window.

# Saving, Restarting, Resetting, and Reinvoking a Simulation

This section tells you how to:

- Save the current state of a simulation in a snapshot.
- Restart a simulation with a saved snapshot.
- Reset a simulation to its original state at time 0.
- Reinvoke a simulation.

## Saving and Restarting the Simulation

You can save and restart the simulation state at any time. Creating simulation checkpoints is especially useful for large simulations where you might want to save the simulation state at regular intervals. Another common use is to save the simulation state after the circuit has been initialized so that future simulations can begin at that point rather than from time 0.

When you save the simulation state, the simulator creates a new snapshot. To restart the simulation at a later time, you must load the saved snapshot.

The current simulation state that is saved in the snapshot includes the simulation time and all object values, scheduled events, annotated delays, the contents of the memory allocated for access type values, and file pointers. It does not include aspects of the debugging environment such as breakpoints, probes, Tcl variables, and GUI configuration. PLI/VPI callbacks and handles are saved under certain circumstances. Please refer to the PLI/VPI manuals for details.

You cannot save a snapshot if the simulator is in the process of executing sequential HDL code. If the simulation is in a state that cannot be saved, you must use the `run -clean` command to run the simulation until the currently running sequential behavior (if any) suspends itself at a delay or event control or a VHDL `wait` statement.

- If you are using the Tcl command-line interface, use the `save` command to save the simulation state and the `restart` command to load a saved snapshot.

See “[save](#)” on page 602 and “[restart](#)” on page 595 for details on these commands. The documentation for the `save` command includes an example of saving and restarting.

- If you are using the SimVision analysis environment, select *File—Simulation Checkpoint*—Save and fill in the Save Simulation form to save the simulation state. To restart, select *File—Simulation Checkpoint—Restart* and fill in the Restart Simulation form. See “[Saving, Restarting, Resetting, and Reinvoking a Simulation](#)” in the *SimVision Analysis Environment User Guide* for an example.

When you restart with a saved snapshot in the same simulation session:

- SHM databases remain open and all probes remain set.
  - Breakpoints set at the time that you execute the restart remain set.
- Note:** If you set a breakpoint that triggers, for example, every 10 ns (that is, at time 10, 20, 30, and so on) and restart with a snapshot saved at time 15, the breakpoint triggers at 20, 30, and so on, not at time 25, 35, and so on.
- Forces and deposits in effect at the time you issue a `save` command are still in effect when you restart.

If you exit the simulation and then invoke the simulator with a saved snapshot, databases are closed. Any probes and breakpoints are deleted. If you want to restore the full Tcl debug environment when you restart, make sure that you save the environment with the `save -environment` command. This command creates a Tcl script that captures the current breakpoints, databases, probes, aliases, and predefined Tcl variable values. You can then use the `Tcl source` command after restarting or the `-input` option when you invoke the simulator to execute the script.

For example:

```
% ncsim top
ncsim> (Open a database, set probes, set breakpoints, deposits, forces, etc.)
ncsim> run 100 ns
ncsim> save worklib.top:ckpt1
ncsim> save -environment ckpt1.tcl
ncsim> exit
% ncsim -tcl worklib.top:ckpt1
ncsim> source ckpt1.tcl
```

The Windows NT and Linux Red Hat (6.1 and 6.2) operating systems impose a two gigabyte limit on the size of a file. If a library database exceeds this limit, you will not be able to add objects to the database. If you save many snapshot checkpoints to unique views in a single library, this file size limit could be exceeded. If you reach this limit, you can:

- Use `save -overwrite` to overwrite an existing snapshot. For example,

```
ncsim> save -simulation -overwrite snap1
```

- Save snapshots to a separate library. For example,

```
% mkdir INCA_libs/snaplib  
% ncsim -f ncsim.args  
ncsim> run 1000 ns  
ncsim> save -simulation snaplib.snap1  
ncsim> run 1000 ns  
ncsim> save -simulation snaplib.snap2
```

- Remove snapshots using the *ncrm* utility. For example,

```
% ncrm -snapshot worklib.snap1
```

## Resetting the Simulation

You can reset the currently loaded model to its original state at time zero.

- If you are using the Tcl command-line interface, use the `reset` command.

See “[reset](#)” on page 594 for details on using the `reset` command.

The documentation for the `save` command includes an example of resetting the simulation. See “[save](#)” on page 602.

- If you are using the SimVision analysis environment, select *File—Reset Simulation*. The time-zero snapshot, created by the elaborator, must still be available.

See “[Saving, Restarting, Resetting, and Reinvoking a Simulation](#)” in the *SimVision Analysis Environment User Guide* for an example of how to save a simulation snapshot and how to restart and reset the simulation.

When you reset the simulation to its state at time 0, the debug environment remains the same.

- Tcl variables remain as they were before the reset.
- SHM and VCD databases remain open, and probes remain set.

**Note:** VCD databases created with the `$dumpvars` call in Verilog source code are closed when you reset.

- Breakpoints remain set.
- Watch Windows and the SimVision Waveform Viewer window remain the same.

Forces and deposits in effect at the time you issue the `reset` command are removed.

If you exit the simulation instead of resetting, databases are closed, probes and breakpoints are deleted, and Tcl variables are reset to their default values.

## Reinvoking a Simulation

If you are using the SimVision analysis environment, you can reinvoke the simulation session at any time, even after the simulation has finished. When you reinvoke, an `exit` command is issued and then *ncsim* is invoked with the `-update` option, which recompiles any changed design units, re-elaborates the design, generates a new snapshot, and loads the updated snapshot.

**Note:** On Windows, a reinvoke causes SimControl to exit before the simulator exits. If you are running the simulator using run scripts or a makefile, subsequent commands in the script or makefile may get executed before the simulator is reinvoked. For example, if your makefile includes commands to clean up work libraries, these commands could get executed before the simulator is reinvoked. This can cause the reinvoke to fail.

To reinvoke, select *File—Reinvoke Simulation*.

If you have set the *Prompt before reinvoke* option on the Preferences form, SimControl displays the Reinvoke form, which lists the command-line arguments you used when you invoked *ncsim* originally. The `-update` option is included in the list. If you want to change the *ncsim* arguments, edit the text field. Click *Yes* to reinvoke the simulation. Click *No* to cancel the reinvoke and remain in the current simulation session.

If the *Prompt before reinvoke* option on the Preferences form is not set, the Reinvoke form does not appear, and *ncsim* is invoked with `-update` and the same original command-line arguments.

Reinvoke works the same way if you are running the NC-Verilog simulator with *ncverilog*. If the *Prompt before reinvoke* option on the Preferences form is set, the Reinvoke form contains the original *ncverilog* command-line arguments. If you edit the text field to add an option that affects elaboration (`+access+`, for example), the design is re-elaborated and a new snapshot is created. See [Chapter 4, “Running NC-Verilog with the ncverilog Command,”](#) for details on using *ncverilog*.

**Note:** Reinvoke is a SimControl option. You can reinvoke the simulation inside the graphical environment as many times as you want, but if you edit the text field on the Reinvoke form to remove the `-gui` option, it is not possible to reinvoke from the Tcl *ncsim>* prompt.

When you reinvoke, your current setup is automatically saved and restored. This includes:

- SHM databases
- Probes
- Watch Windows

## **NC-Verilog Simulator Help**

### Simulating Your Design with ncsim

---

- Breakpoints
- All signals that were displayed in SimVision Waveform Viewer

## Updating Design Changes When You Invoke the Simulator

When you change design units in the hierarchy, you must recompile them and re-elaborate the design hierarchy.

If you want to update and simulate, use the `-update` option on the `ncsim` command to automatically recompile and re-elaborate all out-of-date design units. This option calls `ncupdate` to recompile any changed design units, re-elaborate the design, and generate a new snapshot. It then invokes the simulator and loads the new snapshot.

If you only want to update the snapshot, run the `ncupdate` utility. See “[ncupdate](#)” on page 928 for details on `ncupdate`.

The purpose of `ncupdate` is to provide quick design change turnaround when you have edited a design unit. The modifications to design units cannot cross file boundaries to modify other files. Do not use `ncupdate` (or `ncsim -update`) after adding a design unit, a source file, or compiler directives to the design. For example, `ncupdate` will not update correctly if you edit a source file to define a new macro, or if you change a design unit in a way that introduces a new cross-file dependency. In these cases, recompile the design with `ncvlog -update`.

Use the `-nosource` option with `-update` if you recompile selected parts of your design and then want to automatically re-elaborate and load the new snapshot into the simulator. For example, suppose you have edited two source files, `first.v` and `second.v`, and you only want to include the changes in `second.v`. Recompile the file `second.v` and then use `ncsim -update -nosource`.

```
% ncsim -update -nosource snapshot
```

You can also use `-nosource` when you have changed one design unit in a file with more than one design unit. You can recompile only the unit you have changed and then use `ncsim -update -nosource` to re-elaborate the design and invoke the simulator.

**1. Recompile the changed module.**

```
% ncupdate -unit [lib.]cell[:view]
```

or

```
% ncvlog -unit [lib.]cell[:view]
```

**2. Use `ncsim -update -nosource`**

```
% ncsim -update -nosource snapshot_name
```

If you make a change to any SDF-related file (the SDF source file, the compiled SDF file, the SDF configuration file, or the SDF command file), and then execute an `ncsim -update`

## NC-Verilog Simulator Help

### Simulating Your Design with ncsim

---

command, the elaborator automatically re-annotates the design using the new, up-to-date files. SDF source files that have changed are automatically recompiled. See [Chapter 17, “SDF Timing Annotation.”](#) for details on SDF annotation.

The following example shows how to use `ncsim -update` to automatically recompile, re-elaborate, and load a new snapshot.

```
;# After editing a design unit, use -update to automatically recompile,
;# re-elaborate, and reinvoke the simulator.
;# The -update option calls ncupdate to recompile the file counter.v, which
;# contains the changed module, m16.
% ncsim -messages -update -tcl board
ncsim: v2.2.(a2): (c) Copyright 1995-1999 Cadence Design Systems, Inc.
Updating snapshot worklib.board:module (SSS), reason: file './counter.v' is newer
than expected.
expected: Mon May  3 15:20:43 1999
actual: Tue Jun  1 08:52:57 1999
Updating: module worklib.m16:module (VST)
file: ./counter.v
  module worklib.m16:module
    errors: 0, warnings: 0
;# ncupdate re-elaborates the design hierarchy.
Updating: snapshot worklib.board:module (SSS)
Update of snapshot worklib.board:behav (SSS) successful.
;# The -update option automatically reinvookes the simulator.
Loading snapshot worklib.board:module ..... Done
ncsim>
```

The following example shows how to recompile one module in a file containing multiple modules before using `-update`.

```
;# The source file 16bit_alu.v contains four modules. After editing module
;# logic, use ncupdate-unit to update only that module.
% ncupdate -messages -unit worklib.logic:module
ncupdate: v1.0.(b8): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
Updating: module worklib.logic:module (VST)
file: ./16bit_alu.v
  module worklib.logic:module
    errors: 0, warnings: 0
;# Then use ncsim -update -nosource to re-elaborate, generate a new snapshot,
;# and reinvoke the simulator.
% ncsim -messages -tcl -update -nosource alu_16
ncsim: v1.0.(b8): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
Updating snapshot worklib.alu_16:module (SSS), reason: dependent module
worklib.logic:module (VST) is newer than expected.
```

## **NC-Verilog Simulator Help**

### Simulating Your Design with ncsim

---

```
expected: Tue Oct  8 11:19:04 1996
actual:   Tue Oct  8 11:20:31 1996
Updating: snapshot worklib.alu_16:module (SSS)
Update of snapshot worklib.alu_16:module (SSS) successful.
Loading snapshot worklib.alu_16:module ..... Done
ncsim: *W,VSCNEW: file './16bit_alu.v' is newer than expected by module
worklib.alu_16:module (VST).
ncsim>
```

## Providing Interactive Commands from a File

You can load a file containing simulator commands by specifying an input file. This is useful when you want to load a file of Tcl commands or aliases, or when you want to reproduce an interactive session by re-executing a file of commands saved from a previous simulation run (a key file). When *ncsim* has processed all of the commands in the input file, or if you interrupt processing, input reverts back to the terminal.

There are three ways to execute the commands in an input file:

- Specify the input file with the `-input` option when you invoke the simulator. When you use the `-input` option, commands contained in the input file are executed at the beginning of the simulation session.  
See [-input](#) for information on the `-input` command-line option.
- Specify the input file with the `input` command.  
See [“input”](#) on page 560 for details on the `input` command.
- Execute the Tcl `source` command.
  - If you are using the Tcl command-line interface, enter the command at the prompt. See [“source”](#) on page 616.
  - If you are using the SimVision analysis environment, select *File–Input Commands* in the SimControl window. See [“Executing a File Containing Simulator Commands”](#) in the *SimVision Analysis Environment User Guide* for an example.

The behavior of the `input` command and of the `-input` option is different from the behavior of the `source` command in the following ways:

- With the `source` command, execution of the commands in the script stops if a command generates an error. With the `input` command or with the `-input` option, the contents of the file are read in place of standard input at the next Tcl prompt, as if you had typed the commands at the command-line prompt. This means that errors do not stop the execution of commands in the script.
- The `input` command and the `-input` option echo commands to the screen as they are executed, along with any command output or error messages. The `source` command, on the other hand, displays the output of only the last command in the file. Output from the model (for example, the output of `$display`, `$monitor`, or `$strobe` tasks, or the output of stop points) is printed to the screen.

The following section provides more information on the `-input` command-line option.

## -input Command Syntax

The `-input` option allows you to specify a file name or a simulator command as an argument.

Use the `-input` option with a file name to specify a file of commands.

Example:

```
% ncsim -input setup.inp moda
```

To specify a simulator command, use the `@` symbol before the command, enclosing everything in quotation marks if the command takes an argument, modifier, or option.

Syntax:

```
% ncsim -input @command snapshot
```

Examples:

```
% ncsim -input @run top
```

```
% ncsim -input "@stop -line 22" top
```

You can include more than one `-input` option of either form on the command line. The input files (or commands) are processed in the order in which they appear on the command line.

The following example shows you how to use the `-input` option when you invoke the simulator.

```
;# Create the input file using a text editor. Key files can also be used as
;# input files.
;* command input file: set_break.inp
stop -create -line 27
run
value data
run 50
value data
run 50
value data
run
;# Run ncsim with the -input option to specify the input file. The simulator
;# executes the commands in the file.
% ncsim -messages -input set_break.inp harddrive
ncsim: v1.0.(p2): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
Loading snapshot worklib.harddrive:module ..... Done
ncsim> stop -create -line 27
Created stop 1
```

## NC-Verilog Simulator Help

### Simulating Your Design with ncsim

---

```
ncsim> run
0 FS + 0 (stop 1: ./harddrive.v:27)
./harddrive.v:27      repeat (2)
ncsim> value data
4'hx
ncsim> run 50
Ran until 50 NS + 0
ncsim> value data
4'h0
ncsim> run 50
at time 50 clr =1 data= 0 q= x
Ran until 100 NS + 0
ncsim> value data
4'h0
ncsim> run
at time 150 clr =1 data= 1 q= 0
at time 250 clr =1 data= 2 q= 1
...
...
...
at time 3250 clr =0 data=15 q= 0
at time 3350 clr =0 data=15 q= 0
Simulation complete via $finish(1) at time 3400 NS + 0
;# Control reverts back to the terminal after ncsim has executed all commands
;# in the file.
ncsim> exit
%
```

## Exiting the Simulation

To exit the simulator:

- If you are using the Tcl command-line interface, use either the `exit` command or the `finish` command.

The `exit` command is a built-in Tcl command. It halts execution and returns control to the operating system. See “[exit](#)” on page 548 for details.

The `finish` command also halts execution and returns control to the operating system. This command takes an optional argument that determines what type of information is displayed after exiting.

- 0—Prints nothing (same as executing `finish` without an argument).
- 1—Prints the simulation time.
- 2—Prints simulation time and statistics on memory and CPU usage.

See “[finish](#)” on page 549 for details on the `finish` command.

- If you are using the SimVision analysis environment, select *File–Exit* or enter the `finish` command in the I/O region of the SimControl window.

**Note:** If you type the `finish` command in the SimControl window, the window disappears before you can read the information. However, this information appears in the log file.

## Mixed Verilog/VHDL Simulation

---

The Cadence® NC-Sim mixed-language simulator contains all of the capabilities of the NC-Verilog simulator and the NC-VHDL simulator within a single tool so that you can simulate Verilog, VHDL, or mixed-language designs.

This chapter explains how to import a Verilog module into a VHDL design unit and how to import a VHDL design unit into a Verilog module. Two additional sections present use models for the following special situations:

- Importing a legacy Verilog-XL design into VHDL for simulation with NC-Sim.
- Preparing a Leapfrog Verilog Model Import design for simulation with NC-Sim.

This chapter discusses the following topics:

- [Mapping of Data Types](#)
- [Importing Verilog into VHDL](#)
  - [Using Default Binding](#)
  - [Using a Configuration Specification or Configuration Declaration](#)
  - [Using Direct Instantiation](#)
  - [Using a Shell](#)
- [Importing VHDL into Verilog](#)
  - [Importing VHDL into Verilog without a Shell](#)
  - [Importing VHDL into Verilog with a Shell](#)
  - [Importing VHDL into Verilog with ncverilog](#)
- [A Verilog-VHDL-Verilog Example](#)
- [Generating a Shell with ncshell](#)
- [Importing a Verilog-XL Design into VHDL](#)

- [Preparing a Leapfrog Verilog Model Import Design](#)
- [Mixed-Language Out-of-Module References](#)
- [Path Names and Mixed-Language Designs](#)
- [SDF Annotation for Mixed-Language Designs](#)
- [Generating a Value Change Dump \(VCD\) File for a Mixed-Language Design](#)

## Mapping of Data Types

In a mixed-language design, VHDL signals and generics/generic values may be associated with Verilog ports and parameters, and Verilog nets and parameters/parameter values may be associated with VHDL ports and generics. This section explains the data type conversions that are performed.

### VHDL Generics

The following table shows how VHDL generic types are mapped to Verilog parameters:

<b>VHDL Type</b>	<b>Verilog Parameter</b>
integer	integer
real	real
string	string
time	integer
BOOLEAN	integer
User-defined enumerated types	integer

When passing a Verilog parameter to a VHDL generic, the type that is passed must match the type of the generic, except for VHDL generics of type time, BOOLEAN, and enumerated types. Failure to match the type results in an error at elaboration (for default binding) or at compilation. VHDL generic types cannot override Verilog parameter types. However, if the types match, exact values will be passed. For example, suppose you have a Verilog parameter `x` that is initialized to 3, as follows:

```
parameter x = 3;
```

This parameter can be overridden by a VHDL generic of type integer. If the generic has a value of 5, then `x` will take on the value 5.

**Note:** Verilog parameters cannot be initialized by an expression. For example, in the following case, VHDL generics of type `integer` can be mapped only to parameter `y` and not to `x`.

```
parameter y = 0;
parameter x = 1*y;
```

VHDL generics of type `time` are mapped to Verilog type `integer`. The time unit specified in VHDL is converted to the equivalent in femtoseconds, and then the value is passed to the Verilog parameter. For example, suppose that a Verilog parameter of type `integer` is being overridden by a VHDL generic `G` of type `time`. If the value of `G` is 5 ns, the value passed to the parameter is 5000000. If the value of `G` is 5 ps, the value passed to the parameter is 5000.

Of the predefined enumerated types, only `BOOLEAN` is supported. A generic of type `BOOLEAN` is mapped to a Verilog parameter of type `integer`.

**Note:** The mapping of generics of type `BOOLEAN`, as well as of generics of user-defined enumerated types, is supported only when mixed-language instantiations are done directly without using a shell.

A Verilog parameter of type `integer` that is being overridden by a VHDL generic of type `BOOLEAN` is assigned the value 1 if the generic is `TRUE`, or 0 if the value of the generic is `FALSE`. This is illustrated in the following example. In the output of the `$display` statement, `P1` will have the value 1.

```
-- VHDL design unit
...
...
entity TB is ...
generic G1 : boolean  := TRUE;
...
architecture TB_arch ...

i1 : mod ...    -- Instantiating Verilog module mod
generic map ( P1 => G1 );

end TB_arch;

// Verilog module
module mod(...);
...
parameter P1 = 7;
...
$display(...., P1);    // Output will be 1
...
endmodule;
```

Similarly, a VHDL generic of type `BOOLEAN` that is being overridden by a Verilog parameter with a value of 1 will be assigned the value `TRUE`. If the parameter has a value of 0, the value of the generic will be `FALSE`.

VHDL user-defined enumerated types are also mapped to Verilog parameters of type `integer`. The Verilog parameter will contain the index of the enumerated literal. For example, suppose that you have an enumerated type called `MULTI_LEVEL_LOGIC`, which is defined as follows:

```
type MULTI_LEVEL_LOGIC is (LOW, HIGH, RISING, FALLING, AMBIGUOUS);
```

The following figure shows the above definition and the index numbers for the enumeration literals:

```
type MULTI_LEVEL_LOGIC is (LOW, HIGH, RISING, FALLING, AMBIGUOUS);
INDEX:      0   1   2   3   4
```

Note that, in order for the mapping to work, the value of the Verilog parameter cannot be negative and that it cannot be greater than the index of the last enumeration literal of the enumerated type. In this example, the value of the Verilog parameter can be 0, 1, 2, 3, or 4. Using any other value will cause an error.

In the following example, the value of `P1` in the `$display` statement will be 3, the index for `FALLING`.

```
-- VHDL design unit
...
entity TB is ...
generic G1 : MULTI_LEVEL_LOGIC := FALLING;      -- enumerated type
...
architecture TB_arch ...
...
i1 : mod    -- Instantiating Verilog module mod
generic map ( P1 => G1 );
...
// Verilog module
module mod(...);
...
parameter P1 = 0;
...
$display(...., P1);    // Output will be 3
...
endmodule;
```

Similarly, the value of a Verilog parameter of type integer can be mapped to a VHDL generic of a user-defined enumerated type. The value of the parameter is the index for finding the value of the generic. In the following example, the value of the parameter `x` is 2, so the value of the generic will be RISING.

```
// Verilog module
module top;
    ...
    parameter x = 2;
    ...
    mynand2 #x n1(in1, in2, out);
    ...
endmodule;

-- VHDL design unit
...
...
entity MYNAND2 is
    generic( G : multi_level_logic := HIGH );
    ...
END mynand2;

architecture ... of MYNAND2 is
    ...
end ...;
```

**Note:** When VHDL is instantiated inside a Verilog module, you cannot use a `defparam` statement to assign values to generics defined in the VHDL because a hierarchical path cannot end with a name that refers to a VHDL object or scope. In the previous example, the following `defparam` statement is not allowed because the out-of-module reference terminates in a VHDL object:

```
defparam top.n1.g = 3;
```

See “[Mixed-Language Out-of-Module References](#)” on page 405 for more information.

VHDL generics can have a default value specified in the generic declaration or in the component declaration. You can also specify a value for a generic by mapping a value in the architecture (in the component instantiation), or by mapping a value in the configuration for the component.

If a value is mapped to the generic with a generic map, the default value of the generic is overridden.

- A value passed to a generic with a generic map in a configuration overrides all other default values specified in either the lower level, the component instantiation, or in the component declaration.
- A value passed to a generic with a generic map in a component instantiation overrides a default value specified in the component declaration or in the generic declaration.

This priority order of generic passing is illustrated in the following example. In this example, a VHDL top-level instantiates a VHDL component and Verilog module. A generic of type BOOLEAN called G1 is declared in the entity vhdl\_m. No default value is assigned to the generic.

In the architecture TB\_arch, a default value (`FALSE`) is specified for G1 in the component declarations for component verilog\_m and for component vhdl\_m.

In the component instantiations, generic maps are used to override the default values that were specified in the component declarations. For both instances, the value of G1 is set to `TRUE`.

In the configuration cfg\_TB\_ARCH, the configuration for the Verilog component contains a generic map that specifies a value of `FALSE` for the generic. This overrides the value specified in the lower-level Verilog, in the component instantiation, and in the component declaration.

```
-----  
-- Entity declarations  
-----  
LIBRARY ieee;  
USE ieee.std_logic_1164.all;  
USE ieee.std_logic_arith.all;  
  
entity TB is  
end TB;  
  
LIBRARY ieee;  
USE ieee.std_logic_1164.all;  
USE ieee.std_logic_arith.all;  
  
entity vhdl_m is  
-- Generic declaration. No default value specified.  
    generic (G1 : in boolean);  
    port (a : in std_logic);  
end vhdl_m;  
  
-----  
-- Architecture declaration rtl
```

## NC-Verilog Simulator Help

### Mixed Verilog/VHDL Simulation

---

```
-----
architecture rtl of vhdl_m is

signal G1_local :boolean;

begin
-- Assign the generic to a local signal so the value can be observed.
assign: G1_local <= G1;

end rtl;

-----
-- Architecture declaration TB_arch
-----
architecture TB_arch of TB is

-- Verilog module
component verilog_m
-- Default value of FALSE specified for generic.
-- This value can be overriden in the instantiation or in the configuration.
generic (G1 : boolean := false);
port (a : in std_logic);
end component;

component vhdl_m
-- Default value of FALSE specified for generic.
-- This value can be overriden in the instantiation or in the configuration.
generic (G1 : boolean := false);
port (a : in std_logic);
end component;

signal a_sig : std_logic;

begin

-- Verilog module instantiated
i1 : verilog_m
-- Default value TRUE specified for the generic.
-- This overrides the value specified in the component declaration.
generic map ( G1 => true )
port map (a => a_sig);
```

```
-- VHDL component instantiated
i2 : vhdl_m
-- Default value TRUE specified for the generic.
-- This overrides the value specified in the component declaration.
generic map ( G1 => true)
port map (a => a_sig);

end TB_arch;

-----
-- Configuration
-----
configuration cfg_TB_ARCH of TB is
  for TB_ARCH
  -----
    -- Configure the VHDL lower level
  -----
    for i2: vhdl_m use entity WORK.VHDL_M(RTL);
      for RTL
        end for;
    end for;

  -----
    -- Configure the Verilog lower level
  -----
    for i1: verilog_m use entity work.verilog_m
      -- Default value FALSE passed to the generic.
      -- This overrides all other default values specified in either the lower level,
      -- component instantiation, or component declaration.
      generic map ( G1 => false);
      end for;
    end for;
  end cfg_TB_ARCH;

// File: verilog_f1.v
// Lower-level Verilog module
'timescale 10ps/10ps
module verilog_m(a);

// Lowest priority. Can be overridden by generics from higher level.
parameter G1 = 0;
```

```
input a;

initial
begin : Version_Message
$display("Value of Verilog parameter G1 is:" ,G1);
end
endmodule
```

To simulate this model:

1. Compile the Verilog source file.

```
% ncvlog verilog_f1.v
```

2. Compile the VHDL source file. You must use the -v93 command-line option.

```
% ncvhdl -v93 vhdl_f1.vhd
```

3. Elaborate the design.

```
% ncelab work.cfg_tb_arch -access +r
```

4. Invoke the simulator.

```
% ncsim -tcl work.cfg_tb_arch
ncsim: v03.40.(p001): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
ncsim> run 1 ns
Value of Verilog parameter G1 is: 0
Ran until 1 NS + 0
ncsim> value :i2:g1_local
TRUE
```

## Verilog Parameters

The assigned values of Verilog parameters are carried over as default values for VHDL generics. The following Verilog parameter types are supported:

---

<b>VHDL Type</b>	<b>Verilog Type</b>
integer	integer
real	real
string	string

---

## Port Modes

The following table shows the port modes that are supported in each language and how they are mapped. VHDL ports of mode linkage are not supported.

VHDL	Verilog
in	input
out	output
inout	inout

## VHDL Port Types

The following table shows the VHDL port types that are supported, and the corresponding Verilog types.

VHDL Port Type	Verilog Data Type
std_logic	bit
std_ulogic	bit
std_logic_vector	bit vector
std_ulogic_vector	bit vector
signed	bit vector
unsigned	bit vector

## Verilog Port Types

All the Verilog port types are supported.

## Verilog States

Verilog strengths are mapped to std\_logic and bit as follows:

<b>Verilog</b>	<b>std_logic</b>
HiZ	'Z'
Sm0	'L'
Sm1	'H'
SmX	'W'
Me0	'L'
Me1	'H'
MeX	'W'
We0	'L'
We1	'H'
WeX	'W'
La0	'L'
La1	'H'
LaX	'W'
Pu0	'L'
Pu1	'H'
PuX	'W'
St0	'0'
St1	'1'
StX	'X'
Su0	'0'
Su1	'1'
SuX	'X'

For Verilog states with ambiguous strength:

- std\_logic receives 'X' if either the 0 or 1 strength components are greater than or equal to strong strength.
- std\_logic receives 'W' if both the 0 and 1 strength components are less than strong strength.

VHDL type std\_logic is mapped to Verilog states as follows:

<b>std_logic</b>	<b>Verilog</b>
'U'	StX
'X'	StX
'0'	St0
'1'	St1
'Z'	HiZ
'W'	PuX
'L'	Pu0
'H'	Pu1
'_'	StX

## Importing Verilog into VHDL

You can import a Verilog design unit into VHDL if the Verilog design unit meets the following criteria:

- The Verilog block must be a module. You cannot import a UDP or a macromodule.
- The ports cannot be connected to bidirectional tran gates.

There are a couple of other restrictions and things that you should be aware of:

- You cannot associate a foreign formal, such as a Verilog port, with a type converted VHDL actual (signal) in a port map clause. This is not supported.
- If you are importing a Verilog module into VHDL using direct instantiation, a formal in a port or generic map aspect can only be a simple identifier. Named connections, bit selects, and part selects are not allowed. For example, the following port map will result in an error if you are importing the Verilog module using direct instantiation.

```
U1: entity worklib.bar(module)
      port map (c(4) => cx, c(2 to 3) => dx);
```

See “[Using Direct Instantiation](#)” on page 366 for details on direct instantiation.

- Out-of-module references (OOMR) from Verilog modules must terminate in a Verilog module. The OOMR can go through VHDL hierarchies, but cannot reference a VHDL signal. In other words, a hierarchical path in a Verilog model must end with a name that refers to a Verilog object or scope.

OOMRs are not supported for VHDL design units.

See “[Mixed-Language Out-of-Module References](#)” on page 405 for more information.

- For SDF annotation, interconnect delays, including multi-source interconnect delays, are supported across the language boundary except if the language boundary is a bidirect.

See [Chapter 17, “SDF Timing Annotation”](#) for details on SDF annotation.

- The elaborator, by default, marks all simulation objects in the design as having no read, write, or connectivity access. Turning off these three forms of access allows the elaborator to perform a set of optimizations that can dramatically improve simulation performance. However, this means that, by default, you will not be able to access simulation objects from a point outside the HDL code, through Tcl commands, or through PLI/VPI/VHPI. Access to simulation objects must be explicitly turned on by using elaborator command-line options. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details.

You can import a Verilog module into VHDL by:

■ Using default binding.

In default binding, you write a component declaration for the module and then instantiate the component with a component instantiation statement in which the instantiated unit consists of the name of the component. For example,

```
architecture A of processor is
  component vlog_alu
    port( ... );
  end component;

  begin
    U1 : vlog_alu port map(...); -- Component instantiation statement
  end A;
```

See “[Using Default Binding](#)” on page 353 for more information on importing a Verilog module using default binding.

■ Using a configuration specification or configuration declaration.

If the component that you are importing is a Verilog module, you can use a configuration specification to explicitly bind the Verilog module to a VHDL component instance. Instead of specifying the entity and (optionally) the architecture, you specify the Verilog module name and (optionally) the view. For example:

```
architecture A of processor is
  component vlog_alu
    port( ... );
  end component;

  for all : vlog_alu use entity work.vlog_alu(rtl) [port_map_aspect];

  begin
    U1: vlog_alu port map(...);
  end A;
```

You can also specify the binding in a configuration declaration.

See “[Using a Configuration Specification or Configuration Declaration](#)” on page 358 for more information on importing a Verilog module using a configuration.

■ Using direct instantiation.

You can directly instantiate a Verilog module by using the same type of instantiation statement that you use to directly instantiate a VHDL entity.

For example, if you are instantiating a VHDL design entity, the instantiation statement might look like the following:

```
U1 : entity work.ent(arch) port map (...);
```

If you are importing a Verilog design unit, you would specify the module name followed by (optionally) the view name.

```
U1 : entity work.vlog_alu(rtl) port map (...);
```

This instantiation statement specifies that the instantiation of U1 binds to the Verilog design unit vlog\_alu:rtl present in the library work.

Direct instantiation is a VHDL-93 feature. You must compile the VHDL source files with the -v93 option.

See “[Using Direct Instantiation](#)” on page 366 for more information on importing a Verilog module using direct instantiation.

- Using a shell

You can import Verilog into VHDL by generating a model import shell for the block that you want to import. A VHDL shell contains an entity/architecture pair in which the architecture consists of a `foreign` attribute that points to the compiled Verilog module.

See “[Using a Shell](#)” on page 369 for more information on importing a Verilog module using a VHDL shell.

## Using Default Binding

One way to import a Verilog module into a VHDL design is to write a component declaration for the module and then to instantiate the component with a component instantiation statement in which the instantiated unit is the name of the component. For example,

```
component alu
  generic (....);
  port (....);
end component;

U1: alu port map (...) generic map (...);
```

You can write the component declaration yourself, or you can use the `ncshell` utility to generate it automatically. See “[Using ncshell to Generate the Component Declaration](#)” on page 365 for information on using `ncshell` to generate the component declaration.

Because the Verilog module is bound to the VHDL component instance implicitly when you use default binding, the VHDL component name used in the component declaration must

match the name of the Verilog module. The case of the component name in the component declaration does not have to match the case of the Verilog module name.

The case of the component specified in the instantiation statement does not have to match the case of the Verilog module name.

The names of the ports in the VHDL component declaration must match the names of the ports in the Verilog module declaration. Beginning with version 3.2, the order of the ports in the component declaration can be different from the order of the ports defined in the Verilog module.

When you instantiate the Verilog component, you can map the ports using positional association or named association. For example, suppose that the component `foo` is a Verilog module that is defined as follows:

```
module foo (clk,d,q);
```

The VHDL component is defined as follows:

```
component foo is
    port (CLOCK : in std_logic;
          INPUT : in std_logic;
          OUTPUT : out std_logic);
end component;
```

When you instantiate module `foo` in your VHDL source, both of the following instantiation statements are valid:

```
-- Positional association
u1: foo port map (CLOCK, INPUT, OUTPUT);

-- Named association
u1: foo port map (Q => OUTPUT, CLK => CLOCK, D => INPUT);
```

You can use a mixture of positional and named association.

Because VHDL is case-insensitive, the case of the ports in the VHDL instantiation statement (and in the component declaration) does not have to match the case used in the Verilog.

The implicit binding of Verilog modules to VHDL component instances happens at elaboration time, and all warnings and errors are reported when you elaborate the design with `ncelab`.

Default binding does not impose any compilation order for Verilog and VHDL files.

## Example

In the following example, a VHDL model imports two Verilog modules: module `foo`, which is in a file called `foo.v`, and module `bar`, which is in a file called `bar.v`. The Verilog source files are as follows:

```
// File: foo.v
module foo (x, y, z);
    input x;
    input y;
    output z;

    initial
        $display ("%m, I am module FOO\n");
        assign y = 1'b0;
endmodule

// File: bar.v
module bar(a, b, c);
    input a;
    input b;
    output [1:3] c;

    initial
        $display("%m, I am module BAR\n");
        assign c = 3'b110;
endmodule
```

To import the Verilog modules into VHDL:

1. Write component declarations in the VHDL code for the Verilog modules, and then instantiate the components.

You can write the component declarations manually, or you can use the `ncshell` utility to generate it automatically. See “[Using ncshell to Generate the Component Declaration](#)” on page 365 for information on using `ncshell` to generate the component declaration.

In the instantiation statements, you can use either positional association or named association to map the ports. In the following VHDL model, module `foo` is instantiated three times and module `bar` is instantiated two times. Named association is used to map the ports.

```
-- File: top.vhd
library ieee;
use ieee.std_logic_1164.all;

entity top is
end top;

architecture A of top is
component foo
    port (x: in std_logic;
          y: in std_logic;
          z: out std_logic
        );
end component;

component bar
    port (a: in std_logic;
          b: in std_logic;
          c: out std_logic_vector(2 to 4)
        );
end component;

signal ax: std_logic;
signal bx: std_logic_vector(1 to 5);
signal cx: std_logic;
signal dx: std_logic_vector(1 to 2);
signal ex: std_logic;

begin
    -- The following instantiation associates component
    -- ports with signals of the same width.
    U1: foo port map (z => cx, y => ex, x => ax);

    -- The following instantiation associates a component
    -- port (z) with a slice of an actual signal.
    U2: foo port map (z => bx(4), x => ax, y => ex);

    -- The following instantiation associates a slice of a
    -- component port (c(2 to 3)) with an actual signal.
    -- Note that you can associate mutually exclusive slices of
    -- a Verilog port to different signals.
    U3: bar port map (c(4) => cx, c(2 to 3) => dx, a => ax, b => ex);

    -- The following instantiation associates a component
    -- port (c) with a slice of an actual signal.
```

```
U4: bar port map (c => bx(1 to 3), b => ex, a => ax);  
  
-- The following instantiation associates a component port (y)  
-- with an enumeration literal. Bit_string literals can also  
-- be associated with a port. You must compile with the -v93 option.  
U5: foo port map (z => bx(4), y => '1', x => cx);  
  
ax <= 'Z';  
  
tst_process: process  
begin  
    ax <= '1' after 5 ns;  
    wait;  
end process;  
  
end;
```

2. Compile the Verilog source files using the Verilog compiler (*ncvlog*) and the VHDL source files using the VHDL compiler (*ncvhdl*). You do not have to compile the Verilog files before the VHDL files.

```
% ncvlog foo.v          // Compiles module foo into worklib.foo:module  
% ncvlog bar.v          // Compiles module bar into worklib.bar:module  
% ncvhdl -v93 top.vhd   // Compiles architecture A into WORKLIB.TOP:A
```

3. Elaborate the design using the *ncelab* elaborator. In this example, the *-libverbose* option has been included on the command line to get more detailed binding messages.

```
% ncelab -messages -libverbose worklib.top:a      // Generates snapshot  
                                                // WORKLIB.TOP:A  
ncelab: v3.30.(p1): (c) Copyright 1995 - 2000 Cadence Design Systems, Inc.  
        Elaborating the design hierarchy:  
instance of 'foo' in 'WORKLIB.TOP:A' is resolved to the Verilog module/udp  
worklib.foo:module  
instance of 'foo' in 'WORKLIB.TOP:A' is resolved to the Verilog module/udp  
worklib.foo:module  
instance of 'bar' in 'WORKLIB.TOP:A' is resolved to the Verilog module/udp  
worklib.bar:module  
instance of 'bar' in 'WORKLIB.TOP:A' is resolved to the Verilog module/udp  
worklib.bar:module  
instance of 'foo' in 'WORKLIB.TOP:A' is resolved to the Verilog module/udp  
worklib.foo:module  
        Building instance overlay tables: ..... Done  
        Generating native compiled code:  
...  
...  
Writing initial simulation snapshot: WORKLIB.TOP:A
```

In this example, the elaborator applies default binding rules to bind the Verilog modules `foo` and `bar` to the component instances. The elaborator first searches for a VHDL binding for `foo` and `bar`. If it does not find a VHDL binding, the elaborator searches the entire library structure for an analyzed Verilog module that it can use as a binding, using the following rules:

- a. Look for a Verilog module whose name and case matches that used in the component declaration.
- b. Look for a Verilog module with a matching name, but that is all lowercase.
- c. Look for a Verilog module with a matching name in any case.

If a unique binding is found, and if the case matches, it is used. If a unique binding is found, but the case does not match, it is used with a warning. If multiple bindings are found, the elaborator generates an error message.

In this example, the component names used in the component declarations are `foo` and `bar`, and the Verilog module names are `foo` and `bar`, so the elaborator finds an exact match and uses these modules.

#### 4. Invoke the simulator (*ncsim*) on the simulation snapshot.

```
% ncsim worklib.top:a      // Loads the snapshot into the simulator
```

## Using a Configuration Specification or Configuration Declaration

When you are instantiating a VHDL component, the component declaration can be explicitly bound to a specific entity in the library by using a configuration specification. The binding indication specifies exactly which design unit gets bound to the component instance.

For example, in the following architecture a configuration specification is used to specify that all instances of component `alu` are to be bound to the VHDL entity `ent` and architecture `arch`, which are present in the library `WORK`.

```
architecture A of processor is
  component alu
    port( ... ) ;
  end component;

  for all : alu use entity WORK.ent(arch) [port_map_aspect]; -- Configuration
                                         -- specification

  begin
    U1: alu port map(...);      -- Component instantiation statement
  end A;
```

If the component that you are importing is a Verilog module, you can use a configuration specification to explicitly bind the Verilog module to a VHDL component instance. Instead of specifying the entity and (optionally) the architecture, you specify the Verilog module name and (optionally) the view.

For example, suppose that the component `alu` in the example above is a Verilog module called `alu`. The following configuration specification specifies that all instances of the component `alu` are to be bound to the Verilog design unit `worklib.alu:module`.

```
for all : alu use entity worklib.alu(module) [port_map_aspect];
```

Unlike the case with default binding, when using a configuration specification, the name of the VHDL component in the component declaration, and the names of the component ports, can be different from the Verilog module name and port names.

To ensure that you are binding to exactly the Verilog module/view that you want to bind to, the case used in the module(view) pair in the configuration should match the case of the compiled Verilog design unit. For example, if the compiled Verilog unit is `worklib.ALU:rtl`, the configuration specification should be:

```
for all : alu use entity worklib.ALU(rtl) [port_map_aspect];
```

In the instantiation statements, you can use either positional association or named association to map the ports and generics.

Component/port/generic binding to Verilog happens during parsing, and so all warnings and errors are reported when you compile the source files.

Because the Verilog modules are being configured from within a VHDL source file, the VHDL source file is dependent on the Verilog source files. Therefore, you must compile the Verilog files before you compile the VHDL instantiating unit.

## Example 1

In the following example, a VHDL model imports two Verilog modules: `FOO` and `BAR`. Module `FOO` has an RTL view, which is described in the file `foo.v`, and a gate-level view, which is described in the file `foo.vg`. Module `BAR` is described in a file called `bar.v`. The architecture includes configuration specifications to control the binding.

The steps for simulating this example are as follows:

1. Compile the Verilog source code for the Verilog block that you want to import using the Verilog compiler (`ncvlog`).

**Note:** Because the Verilog modules are being configured from within a VHDL source file, the VHDL source file is dependent on the Verilog source files. Therefore, you must

compile the Verilog files before you compile the VHDL instantiating unit.

```
% ncvlog foo.v -view rtl      // Compiles module FOO into worklib.FOO:rtl  
% ncvlog foo.vg -view gate    // Compiles module FOO into worklib.FOO:gate  
% ncvlog bar.v                // Compiles module BAR into worklib.BAR:module
```

2. Write a component declaration in the VHDL code for the Verilog modules, and then instantiate the components. You can write the component declaration manually, or you can use the *ncshell* utility to generate it automatically. See [“Using ncshell to Generate the Component Declaration”](#) on page 365 for information on using *ncshell* to generate the component declaration.

In this example, there are two components called `vlog_foo` and `vlog_bar`. Notice that the name of the components can be different from the name of the Verilog modules.

Now write the configuration specifications to specify the binding explicitly.

```
-- File: top.vhd  
library ieee;  
use ieee.std_logic_1164.all;  
library worklib;  
  
entity top is  
end top;  
  
architecture A of top is  
component vlog_foo  
port (x: in std_logic;  
      y: in std_logic;  
      z: out std_logic  
);  
end component;  
  
component vlog_bar  
port (a: in std_logic;  
      b: in std_logic;  
      c: out std_logic_vector(2 to 4)  
);  
end component;  
  
signal ax: std_logic;  
signal bx: std_logic_vector(1 to 5);  
signal cx: std_logic;  
signal dx: std_logic_vector(1 to 2);  
signal ex: std_logic;
```

```
-- Configuration specifications
for U1 : vlog_foo use entity worklib.FOO(gate);
for others : vlog_foo use entity worklib.FOO(rtl);
for all : vlog_bar use entity worklib.bar(module);

begin
-- Component instantiation statements
  U1: vlog_foo port map (z => cx, y => ex, x => ax);
  U2: vlog_foo port map (z => bx(4), x => ax, y => ex);
  U3: vlog_bar port map (c(4) => cx, c(2 to 3) => dx, a => ax, b => ex);
  U4: vlog_bar port map (c => bx(1 to 3), b => ex, a => ax);
  U5: vlog_foo port map (z => bx(4), y => '1', x => cx);

  ax <= 'Z';

  tst_process: process
begin
  ax <= '1' after 5 ns;
  wait;
end process;

end;
```

**3. Compile the VHDL source code using *ncvhdl*.**

```
% ncvhdl -v93 top.vhd      // Compiles architecture A into WORKLIB.TOP:A
```

**4. Elaborate the design using the *ncelab* elaborator. In this example, the *-libverbose* option has been included on the command line to get more detailed binding messages.**

```
% ncelab -messages -libverbose worklib.top:a      // Generates snapshot
                                                // WORKLIB.TOP:A
```

The following output shows the output of *ncvlog*, *ncvhdl*, and *ncelab*:

```
ncvlog: v03.30.(p001): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
file: foo.v
      module worklib.FOO:rtl
          errors: 0, warnings: 0
ncvlog: v03.30.(p001): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
file: foo.vg
      module worklib.FOO:gate
          errors: 0, warnings: 0
```

## NC-Verilog Simulator Help

### Mixed Verilog/VHDL Simulation

---

```
ncvlog: v03.30.(p001): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
file: bar.v
    module worklib.BAR
        errors: 0, warnings: 0
ncvhdl: v03.30.(p001): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
top.vhd:
    for all : vlog_bar use entity worklib.bar(module);
|
ncvhdl_p: *W,VLCINM (top.vhd,31|45): Verilog Unit Bound: (worklib.BAR:module)
does not match Exact CASE.
    errors: 0, warnings: 1
WORKLIB.TOP (entity):
    streams: 1, words: 3
WORKLIB.TOP:A (architecture):
    streams: 4, words: 227
ncelab: v03.30.(p001): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
    Elaborating the design hierarchy:
Resolved module/instance 'vlog_foo' at ':top(A):U1' to 'worklib.FOO:gate'
Resolved module/instance 'vlog_foo' at ':top(A):U2' to 'worklib.FOO:rtl'
Resolved module/instance 'vlog_bar' at ':top(A):U3' to 'worklib.BAR:module'
Resolved module/instance 'vlog_bar' at ':top(A):U4' to 'worklib.BAR:module'
Resolved module/instance 'vlog_foo' at ':top(A):U5' to 'worklib.FOO:rtl'
    Building instance overlay tables: ..... Done
    Generating native compiled code:
    ...
    ...
    Writing initial simulation snapshot: WORKLIB.TOP:A
%
```

The parser first searches for a VHDL binding for FOO and BAR. If it does not find a VHDL binding, the parser searches the entire library structure for an analyzed Verilog module that it can use as a binding, using the following rules:

- a.** Look for a Verilog module whose name and case matches that used in the binding indication of the configuration specification.
- b.** Look for a Verilog module with a matching name, but that is all lowercase.
- c.** Look for a Verilog module with a matching name in any case.

If a unique binding is found, and if the case matches, it is used. If a unique binding is found, but the case does not match, it is used with a warning. If multiple possible bindings are found, the parser generates an error message.

In this example, the parser finds exact matches for `FOO:gate` and for `FOO:rtl`. However, the parser generates a `VLCINM` warning message because the Verilog module name is `BAR`, but the configuration specification uses lowercase.

Specifying a view name in the configuration specification is optional. If you do not specify a view name, and if there are multiple views in the library for a module, the parser generates an error.

In this example, the parser would generate an error if the configuration specifications were as follows because there are two views (`rtl` and `gate`) of module `FOO` in the library `worklib`.

```
for U1 : vlog_foo use entity worklib.FOO;  
for others : vlog_foo use entity worklib.FOO;
```

**5. Invoke the simulator (`ncsim`) on the simulation snapshot.**

```
% ncsim worklib.top:a           // Loads the snapshot into the simulator
```

## Example 2

In the previous example, configuration specifications contained in the same architecture as the component declarations were used to specify binding. You can also use a configuration declaration to specify the binding. For example, instead of using the configuration specifications shown in the example above, you could write the following configuration file:

```
-- File vhdl_conf.vhd  
configuration CONF of TOP is  
    for A  
        for U1 : vlog_foo use entity worklib.FOO(rtl);  
        end for;  
        for others : vlog_foo use entity worklib.FOO(gate);  
        end for;  
        for all : vlog_bar use entity worklib.bar(module);  
        end for;  
    end for;  
end configuration CONF;
```

The steps for simulating this example are as follows:

**1. Compile the Verilog source code for the Verilog block that you want to import using the Verilog compiler (`ncvlog`).**

**Note:** Because the Verilog modules are being configured from within a VHDL source file, the VHDL source file is dependent on the Verilog source files. Therefore, you must compile the Verilog files before you compile the VHDL instantiating unit.

## NC-Verilog Simulator Help

### Mixed Verilog/VHDL Simulation

---

```
% ncvlog foo.v -view rtl      // Compiles module FOO into worklib.FOO:rtl
% ncvlog foo.vg -view gate   // Compiles module FOO into worklib.FOO:gate
% ncvlog bar.v               // Compiles module BAR into worklib.BAR:module
```

2. Write a component declaration in the VHDL code for the Verilog modules, and then instantiate the components.

3. Compile the VHDL source code using *ncvhdl*.

```
% ncvhdl -v93 top.vhd        // Compiles architecture A into WORKLIB.TOP:A
% ncvhdl vhdl_conf.vhd       // Compiles configuration CONF into
                             // WORKLIB.CONF:CONFIGURATION
```

4. Elaborate the design using the *ncelab* elaborator. In this example, the *-libverbose* option has been included on the command line to get more detailed binding messages.

```
% ncelab -messages -libverbose worklib.conf      // Generates snapshot
                                                // WORKLIB.CONF
```

The following output shows the output of *ncvlog*, *ncvhdl*, and *ncelab*:

```
ncvlog: v03.30.(p001): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
file: foo.v
      module worklib.FOO:rtl
          errors: 0, warnings: 0
ncvlog: v03.30.(p001): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
file: foo.vg
      module worklib.FOO:gate
          errors: 0, warnings: 0
ncvlog: v03.30.(p001): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
file: bar.v
      module worklib.BAR
          errors: 0, warnings: 0
ncvhdl: v03.30.(p001): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
top.vhd:
      errors: 0, warnings: 0
WORKLIB.TOP (entity):
      streams: 1, words: 3
WORKLIB.TOP:A (architecture):
      streams: 1, words: 197
ncvhdl: v03.30.(p001): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
vhdl_conf.vhd:
      for all : vlog_bar use entity worklib.bar(module);
      |
ncvhdl_p: *W,VLCINM (vhdl_conf.vhd,9|41): Verilog Unit Bound:
(worklib.BAR:module) does not match Exact CASE.
      errors: 0, warnings: 1
```

```
WORKLIB.CONF (configuration):
    streams: 4, words: 33
ncelab: v03.30.(p001): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
        Elaborating the design hierarchy:
Resolved module/instance 'vlog_foo' at ':top(A):U1' to 'worklib.FOO:rtl'
Resolved module/instance 'vlog_foo' at ':top(A):U2' to 'worklib.FOO:gate'
Resolved module/instance 'vlog_bar' at ':top(A):U3' to 'worklib.BAR:module'
Resolved module/instance 'vlog_bar' at ':top(A):U4' to 'worklib.BAR:module'
Resolved module/instance 'vlog_foo' at ':top(A):U5' to 'worklib.FOO:gate'
        Building instance overlay tables: ..... Done
Generating native compiled code:
...
...
Writing initial simulation snapshot: WORKLIB.CONF
%
```

## 5. Invoke the simulator (*ncsim*) on the simulation snapshot.

```
% ncsim worklib.conf      // Loads the snapshot into the simulator
```

**Note:** If you have a pure VHDL design, you can automatically generate a configuration file with the *ncelab -conffile* option. You could then edit the configuration file to point to the Verilog module(s) that you want to import. See the description of the *-conffile* option for more information on generating a configuration file.

## Using *ncshell* to Generate the Component Declaration

You can run the *ncshell* utility to automatically generate the component declaration.

First, compile the Verilog source and then run *ncshell*. The argument to the *ncshell* command is the lib.cell:view specification for the compiled module.

```
% ncvlog foo.v bar.v
% ncshell -import verilog -into vhdl worklib.foo:module
% ncshell -import verilog -into vhdl worklib.bar:module
```

This *ncshell* command generates a component declaration in a file called *module\_name\_comp.vhd*. For example, the component declaration for module *foo* shown below is created in a file called *foo\_comp.vhd*.

```
library ieee;
use ieee.std_logic_1164.all;

package HDLModels is

component foo
    port (
        x: in std_logic;
        y: in std_logic;
        z: out std_logic
    );
end component;

end HDLModels;
```

You can then cut and paste the component declaration (without the enclosing package statements) into your VHDL code, or you can manually compile the package and then include the package in the VHDL so that the component declaration is visible.

By default, *ncshell* escapes uppercase and mixed-case Verilog identifiers in the VHDL shell. For example, if the Verilog module is `vlog`, this identifier appears in the VHDL shell as `\vlog\`. Use the `-noescape` option if you want the Verilog module or port names to be matched exactly in the shell.

See “[Generating a Shell with ncshell](#)” on page 385 for details on *ncshell*.

## Using Direct Instantiation

In VHDL, you can directly instantiate a VHDL design entity using a component instantiation statement, such as the following:

```
U1 : entity WORK.ent(arch) [generic_map_aspect] [port_map_aspect]
```

You can use direct instantiation to explicitly bind Verilog modules to VHDL instances without component declarations. Instead of specifying the entity and (optionally) the architecture, you specify the Verilog module name and (optionally) the view. For example,

```
U1 : entity worklib.alu(rtl) [generic_map_aspect] [port_map_aspect]
```

**Note:** Direct instantiation of a design entity is a VHDL-93 feature. You must compile the VHDL source files with the `-v93` command-line option.

In the instantiation statements, you can use either positional association or named association to map the ports and generics.

If you are importing a Verilog module into VHDL using direct instantiation, a formal in a port or generic map aspect can only be a simple identifier. Named connections, bit selects, and part selects are not allowed. For example, the following port map will result in an error.

```
U1: entity worklib.bar(module)
    port map (c(4) => cx, c(2 to 3) => dx);
```

Component/port/generic binding to Verilog happens during parsing, and so all warnings and errors are reported when you compile the source files.

Because the Verilog modules are being directly instantiated from within a VHDL source file, the VHDL source file is dependent on the Verilog source files. Therefore, you must compile the Verilog files before you compile the VHDL instantiating unit.

## Example

In this example, instance U1 is bound to the Verilog design unit worklib.foo:rtl. Instances U2 and U5 are bound to worklib.foo:gate. Both instances of module bar are bound to worklib.bar:module.

```
library ieee;
use ieee.std_logic_1164.all;
library worklib;

entity top is
end top;

architecture A of top is
signal ax: std_logic;
signal bx: std_logic_vector(1 to 5);
signal cx: std_logic;
signal dx: std_logic_vector(1 to 2);
signal ex: std_logic;

begin
U1 : entity worklib.foo(rtl)
    port map (z => cx, y => ex, x => ax);

U2 : entity worklib.foo(gate)
    port map (z => bx(4), x => ax, y => ex);

U3 : entity worklib.bar(module)
    port map (a => ax, b => ex);
```

```

U4 : entity worklib.bar(module)
      port map (c => bx(1 to 3), b => ex, a => ax);

U5 : entity worklib.foo(gate)
      port map (z => bx(4), y => '1', x => cx); --allowed with -v93 option only.
ax <= 'Z';

tst_process: process
begin
  ax <= '1' after 5 ns;
  wait;
end process;

end;

```

To simulate this example:

1. Compile the Verilog source code for the Verilog block that you want to import using the Verilog compiler (*ncvlog*).

**Note:** Because the Verilog modules are being configured from within a VHDL source file, the VHDL source file is dependent on the Verilog source files. Therefore, you must compile the Verilog files before you compile the VHDL instantiating unit.

```
% ncvlog foo.v -view rtl      // Compiles module FOO into worklib.FOO:rtl
% ncvlog foo.vg -view gate   // Compiles module FOO into worklib.FOO:gate
% ncvlog bar.v               // Compiles module BAR into worklib.BAR:module
```

2. Compile the VHDL source code using *ncvhdl*.

```
% ncvhdl -v93 top.vhd        // Compiles architecture A into WORKLIB.TOP:A
```

3. Elaborate the design using the *ncelab* elaborator. In this example, the *-libverbose* option has been included on the command line to get more detailed binding messages.

```
% ncelab -messages -libverbose worklib.top:a      // Generates snapshot
                                                // WORKLIB.TO:A
ncelab: v03.30.(p001): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
      Elaborating the design hierarchy:
Resolved module/instance 'foo' at '::top(A):U1' to 'worklib.foo:rtl'
Resolved module/instance 'foo' at '::top(A):U2' to 'worklib.foo:gate'
Resolved module/instance 'bar' at '::top(A):U3' to 'worklib.bar:module'
Resolved module/instance 'bar' at '::top(A):U4' to 'worklib.bar:module'
Resolved module/instance 'foo' at '::top(A):U5' to 'worklib.foo:gate'
      Building instance overlay tables: ..... Done
      Generating native compiled code:
      ...

```

```
...  
Writing initial simulation snapshot: WORKLIB.TOP:A  
%
```

**4. Invoke the simulator (*ncsim*) on the simulation snapshot.**

```
% ncsim worklib.top:a      // Loads the snapshot into the simulator
```

When you elaborate this design, the elaborator first searches for a VHDL binding for `foo` and `bar`. If it does not find a VHDL binding, the elaborator searches the entire library structure for an analyzed Verilog module that it can use as a binding, using the following rules:

- a. Look for a Verilog module whose name and case matches that used in the instantiation statement.
- b. Look for a Verilog module with a matching name, but that is all lowercase.
- c. Look for a Verilog module with a matching name in any case.

If a unique binding is found, and if the case matches, it is used. If a unique binding is found, but the case does not match, it is used with a warning. If multiple possible bindings are found, the elaborator generates an error message.

Specifying a view name in the instantiation statement is optional. If you do not specify a view name, and if there are multiple views in the library for a module, the elaborator generates an error.

## Using a Shell

You can always generate and use a model shell to import a Verilog module into VHDL.

The example used in this section is the same example used in “[Using Default Binding](#)” on page 353, in which two Verilog modules (module `foo` and module `bar`) are imported into a top-level VHDL model.

To import the Verilog modules into VHDL using a shell:

1. Compile the Verilog source code for the Verilog blocks that you want to import using the Verilog compiler (*ncvlog*).

```
% ncvlog foo.v      // Compiles module foo into worklib.foo:module  
% ncvlog bar.v      // Compiles module bar into worklib.bar:module
```

2. Generate a model import shell for the block that you want to import using the *ncshell* utility. The argument to the *ncshell* command is the lib.cell:view specification for the compiled module.

## NC-Verilog Simulator Help

### Mixed Verilog/VHDL Simulation

---

```
% ncshell -import verilog -into vhdl worklib.foo:module
% ncshell -import verilog -into vhdl worklib.bar:module
```

The ncshell command generates a VHDL shell in a file called *module\_name.vhd* and then compiles the file. The two shell files for this example (*foo.vhd* and *bar.vhd*) are as follows:

```
-- File foo.vhd
library ieee;
use ieee.std_logic_1164.all;

entity foo is
  port (
    x: in std_logic;
    y: in std_logic;
    z: out std_logic
  );
end foo;

architecture verilog of foo is
  attribute foreign of verilog:architecture is "VERILOG(event)
                                             worklib.foo:module";
begin
end;

-- File bar.vhd
library ieee;
use ieee.std_logic_1164.all;

entity bar is
  port (
    a: in std_logic;
    b: in std_logic;
    c: out std_logic_vector(1 to 3)
  );
end bar;

architecture verilog of bar is
  attribute foreign of verilog:architecture is "VERILOG(event)
                                             worklib.bar:module";
begin
end;
```

Notice that the name of the architecture in the shell defaults to *verilog*.

**Note:** In Verilog, identifiers are case-sensitive. By default, mixed-case and uppercase identifiers in Verilog are escaped in VHDL shells. For example, if the Verilog module is Vlog, this identifier appears in the VHDL shell as \Vlog\. Use the `-noescape` option if you want the Verilog module name to be matched exactly in the shell. Do not set the `CDS_ALT_NMP` environment variable. This variable is not supported.

3. In the VHDL file, specify that architecture `verilog` is to be used for entity `foo` and for entity `bar`. The VHDL file instantiating the Verilog modules in this example is as follows:

```
-- File: top.vhd
library ieee;
use ieee.std_logic_1164.all;
library worklib;

entity top is
end top;

architecture A of top is

component foo
    port (x: in std_logic;
          y: in std_logic;
          z: out std_logic
        );
end component;

component bar
    port (a: in std_logic;
          b: in std_logic;
          c: out std_logic_vector(2 to 4)
        );
end component;

for all: foo use entity worklib.foo(verilog);
for all: bar use entity worklib.bar(verilog);

signal ax: std_logic;
signal bx: std_logic_vector(1 to 5);
signal cx: std_logic;
signal dx: std_logic_vector(1 to 2);
signal ex: std_logic;
```

```
begin
i1: foo port map (z => cx, y => ex, x => ax);
i2: foo port map (z => bx(4), x => ax, y => ex);
i3: bar port map (c(4) => cx, c(2 to 3) => dx, a => ax, b => ex);
i4: bar port map (c => bx(1 to 3), b => ex, a => ax);
i5: foo port map (z => bx(4), y => '1', x => cx);
ax <= 'Z';

tst_process: process
begin
    ax <= '1' after 5 ns;
    wait;
end process;

end;
```

**4. Compile the top-level VHDL file (top.vhd).**

```
% ncvhdl -v93 top.vhd      // Compiles architecture A into WORKLIB.TOP:A
```

**5. Elaborate the design with *ncelab*.**

```
% ncelab worklib.top:a      // Generates snapshot WORKLIB.TOP:A
```

**6. Invoke the simulator (*ncsim*) on the simulation snapshot.**

```
% ncsim worklib.top:a
```

## Importing VHDL into Verilog

You can import a VHDL block into a Verilog module if the VHDL design unit meets the following criteria:

- The design unit is an entity/architecture pair or a configuration declaration.
- The entity ports are of type `std_logic`, `std_ulogic`, `std_logic_vector`, `std_ulogic_vector`, `signed`, or `unsigned`.
- VHDL ports cannot be bidirects that connect to Verilog tran gates.
- The generics are of type `integer`, `positive`, `natural`, `real`, `string`, or `time`. NC-Sim does not support enumerated types for generics.

Because VHDL is case-insensitive, there can never be multiple entities with names like `vhdl1`, `Vhdl1`, or `VHDL`. You can, therefore, use lowercase, uppercase, or mixed-case for the VHDL entity name in the Verilog instantiation.

Out-of-module references (OOMR) from Verilog modules must terminate in a Verilog module. The OOMR can go through VHDL hierarchies, but cannot reference a VHDL signal. In other words, a hierarchical path in a Verilog model must end with a name that refers to a Verilog object or scope.

OOMRs are not supported for VHDL design units. See “[Mixed-Language Out-of-Module References](#)” on page 405 for more information.

For SDF annotation, interconnect delays are not supported across the language boundary. Use port delays instead of interconnect delays. See [Chapter 17, “SDF Timing Annotation”](#) for details on SDF annotation.

Remember that the elaborator, by default, marks all simulation objects in the design as having no read, write, or connectivity access. This means that, by default, you will not be able to access simulation objects from a point outside the HDL code, through Tcl commands, or through PLI/VPI/VHPI. Access to simulation objects must be explicitly turned on using elaborator command-line options. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details.

To reference a VHDL entity or configuration from a Verilog module, you instantiate the VHDL design unit in your Verilog code using the same Verilog language mechanism that you use for instantiating another Verilog module.

When you instantiate the VHDL component, you can map the ports using positional association or named association. For example, suppose that the entity `foo` is a VHDL design unit defined as follows:

```
entity foo is
  port (clk : in std_logic;
        d : in std_logic_vector(1 to 3);
        q : out std_logic_vector(1 to 3)
      );
```

When you instantiate this design unit in your Verilog source, both of the following instantiation statements are valid:

```
// Positional association
foo u1 (clock, input, output);

// Named association
foo u2 (.q(output), .clk(clock), .d(input));
```

When instantiating a VHDL design unit in a Verilog module, you cannot use a mixture of positional and named association. The Verilog LRM states that the two types of module port connections cannot be mixed.

You can also import a VHDL block into a Verilog module by generating a model import shell for the VHDL block. The shell is a Verilog module that contains a `foreign` attribute that points to the compiled VHDL architecture.

Using a Verilog shell to import a VHDL block is required if component names are different from the actual design unit names. For example, suppose that you have a component called `foo` in your Verilog code, but the actual name of the VHDL design unit is `myfoo`. After you generate the model shell, you can specify the correct binding in Verilog for the VHDL design unit by modifying the `foreign` attribute in the shell so that it uses the correct design unit.

```
(*const integer foreign="VHDL(event) library.myfoo:structural";*)
```

The following two examples illustrate how to import VHDL into Verilog. The first example, [“Importing VHDL into Verilog without a Shell”](#) on page 375, shows you how to import a Verilog module without using a shell.

The second example, [“Importing VHDL into Verilog with a Shell”](#) on page 377, shows you how to import a Verilog module by using a model shell.

## Importing VHDL into Verilog without a Shell

In the example shown in this section, a Verilog module imports the following VHDL model.

```
-- File foo.vhd
library ieee;
use ieee.std_logic_1164.all;

entity foo is
    port (a: in std_logic_vector(1 to 3);
          b: out std_logic_vector(1 to 3)
        );
end foo;

architecture foo_arch of foo is
begin
    b <= "010";
end;
```

To import this VHDL into Verilog without using a shell:

1. Compile the VHDL source code for the VHDL block that you want to import using *ncvhdl*.

```
% ncvhdl foo.vhd           // Compiles architecture foo_arch into
                           // WORKLIB.FOO:FOO_ARCH
```

2. Instantiate the VHDL design unit in your Verilog code using the normal Verilog language mechanism for instantiating components.

In the instantiation statements, you can use either positional association or named association to map the ports. In the following Verilog module, the VHDL design unit *foo* is instantiated two times using named association.

```
// File top.v
module top;
    reg [1:3] a1x, a2x;
    wire [1:3] a1x_w = a1x;
    wire [1:4] b1x;
    wire b2x;

    initial
    begin
        a1x = 3'b1Z0;
        a2x = 3'bX11;
    end
```

```
// The following instantiation associates a formal VHDL port (a) with
// the Verilog signal alx_w.
// The VHDL formal port b is associated with a complicated actual expression,
// in this case, a concatenation. The various subparts of the
// concatenation expression can be entire signals or slices of a signal.
foo i1 (.b({b1x[2:3], b2x}), .a(alx_w));

// The following instantiation associates a constant expression (3'b101)
// with a VHDL port (a).
foo i2 (.b({b1x[2:3], b2x}), .a(3'b101));
endmodule
```

**3. Compile your Verilog source files using *ncvlog*.**

```
% ncvlog top.v           // Compiles module top into worklib.top:module
```

**4. Elaborate the design using the *ncelab* elaborator. Include the *-libverbose* option if you want detailed binding messages.**

```
% ncelab worklib.top:module      // Generates snapshot worklib.top:module
ncelab: v3.20.(p1): (c) Copyright 1995 - 2000 Cadence Design Systems, Inc.
```

Elaborating the design hierarchy:

Resolving module/udp 'foo' at 'top.i1'.

Caching library 'worklib' ..... Done

library: 'worklib' views: 'module' 'udp' -> not found

Caching library 'std' ..... Done

library: 'std' views: 'module' 'udp' -> not found

Caching library 'synopsys' ..... Done

library: 'synopsys' views: 'module' 'udp' -> not found

Caching library 'ieee' ..... Done

library: 'ieee' views: 'module' 'udp' -> not found

Caching library 'ambit' ..... Done

library: 'ambit' views: 'module' 'udp' -> not found

instance of module 'foo' in 'worklib.top:module' is resolved to the VHDL architecture WORKLIB.FOO:FOO\_ARCH

Resolving module/udp 'foo' at 'top.i2'.

library: 'worklib' views: 'module' 'udp' -> not found

library: 'std' views: 'module' 'udp' -> not found

library: 'synopsys' views: 'module' 'udp' -> not found

library: 'ieee' views: 'module' 'udp' -> not found

library: 'ambit' views: 'module' 'udp' -> not found

instance of module 'foo' in 'worklib.top:module' is resolved to the VHDL architecture WORKLIB.FOO:FOO\_ARCH

Building instance overlay tables: ..... Done

Generating native compiled code:

...  
...

```
Writing initial simulation snapshot: worklib.top:module
```

The elaborator searches for a Verilog binding for instances `i1` and `i2`. If it does not find a Verilog binding, the elaborator searches the entire library structure for a successful VHDL binding (an analyzed VHDL architecture for entity `foo` or a VHDL configuration for entity `foo`). If a unique binding is found, it is used. If multiple bindings are found, the elaborator generates an error message.

5. Invoke the simulator (`ncsim`) on the simulation snapshot.

```
% ncsim worklib.top:module      // Loads the snapshot into the simulator
```

## Importing VHDL into Verilog with a Shell

This section shows you how to import a VHDL design unit into a Verilog module. The example is the same example used in the previous section.

To import VHDL into Verilog using a shell:

1. Compile the VHDL source code for the VHDL block that you want to import using `ncvhdl`.

```
% ncvhdl foo.vhd          // Compiles architecture foo_arch into  
                           // WORKLIB.FOO:FOO_ARCH
```

2. Generate a model import shell for the block that you want to import using the `ncshell` utility. The argument to the `ncshell` command is the `lib.cell:view` specification for the compiled design unit.

```
% ncshell -import vhdl -into verilog worklib.foo:foo_arch
```

This command generates a Verilog shell in a file called `entity_name.vs`. The shell file for this example (`foo.vs`) is as follows:

```
module foo(  
    a, b  
)  
    (* const integer foreign = "VHDL(event) WORKLIB.foo:foo_arch"; *)  
    input [1:3] a;  
    output [1:3] b;  
endmodule
```

`ncshell` also updates the `hdl.var` file to include or modify the `VIEW_MAP` variable. The `hdl.var` file now includes the following line:

```
define VIEW_MAP ($VIEW_MAP, .vs => shell)
```

If no mapping has already been specified for files with a `.vs` extension, `.vs` files will be compiled with a view name of `shell`.

*ncshell* then compiles the shell file. In this example, `foo.vs` is compiled into `worklib.foo:shell`.

See “[Generating a Shell with ncshell](#)” on page 385 for details on *ncshell*.

**3. Compile your Verilog using *ncvlog*.**

```
% ncvlog top.v          // Compiles module top into worklib.top:module
```

**4. Elaborate the design using the *ncelab* elaborator. Include the `-libverbose` option if you want detailed binding messages.**

```
% ncelab -libverbose worklib.top:module      // Generates snapshot
                                                // worklib.top:module
ncelab: v3.20.(p1): (c) Copyright 1995 - 2000 Cadence Design Systems, Inc.
        Elaborating the design hierarchy:
Resolving module/udp 'foo' at 'top.il'.
        Caching library 'worklib' .... Done
        library: 'worklib' views: 'shell' -> found
Resolved module/udp 'foo' at 'top.il' to 'worklib.foo:shell'.
Module 'worklib.foo:shell' at 'top.il@foo<module>' is a shell module for VHDL
architecture WORKLIB.FOO:FOO_ARCH.
Resolved module/udp 'foo' at 'top.i2' to 'worklib.foo:shell'.
Module 'worklib.foo:shell' at 'top.i2@foo<module>' is a shell module for VHDL
architecture WORKLIB.FOO:FOO_ARCH.
        Building instance overlay tables: ..... Done
        Generating native compiled code:
...
...
Writing initial simulation snapshot: worklib.top:module
```

**5. Invoke the simulator (*ncsim*) on the simulation snapshot.**

```
% ncsim worklib.top:module      // Loads the snapshot into the simulator
```

## **Verilog Shell File Names**

When you import a VHDL model into Verilog using a model import shell, you can specify the file extension for the shell with the `-suffix` option. If you do not use the `-suffix` option *ncshell* gives the shell a `.vs` file extension.

## **Verilog Shell View Names**

*ncshell* invokes *ncvlog* to analyze the Verilog shell in the library where the original VHDL model was analyzed. The following rules determine the view name under which the Verilog shell is analyzed:

1. If the `hdl.var` file has a `VIEW_MAP` variable specified, then that view name mapping is used.
2. If the `hdl.var` file does not have a `VIEW_MAP` defined, then a `VIEW_MAP` variable is added to the `hdl.var` file with the following syntax:

```
define VIEW_MAP ($VIEW_MAP, file_extension_in_use => view_name)
```

  - ❑ If `view_name` is provided through the `-view` option, then it is used.
  - ❑ In all other cases, the default `view_name` of shell is used.

## Importing VHDL into Verilog with ncverilog

Simulating a mixed-language design with *ncverilog* is a two-step process:

1. Compile the VHDL design units that you want to import with *ncvhdl*.

To do this, you must first create a `cds.lib` file to define the libraries that you want to compile the design units into, and an `hdl.var` file to define the work library.

2. Run *ncverilog* with the `+mixedlang` option.

When you run *ncverilog*, the parser searches for a Verilog binding that corresponds to the VHDL instance. If no Verilog binding is found, the elaborator searches the entire library structure for a VHDL binding. If a unique binding is found, it is used. If the elaborator doesn't find a binding, or if multiple bindings are found, the elaborator generates an error message.

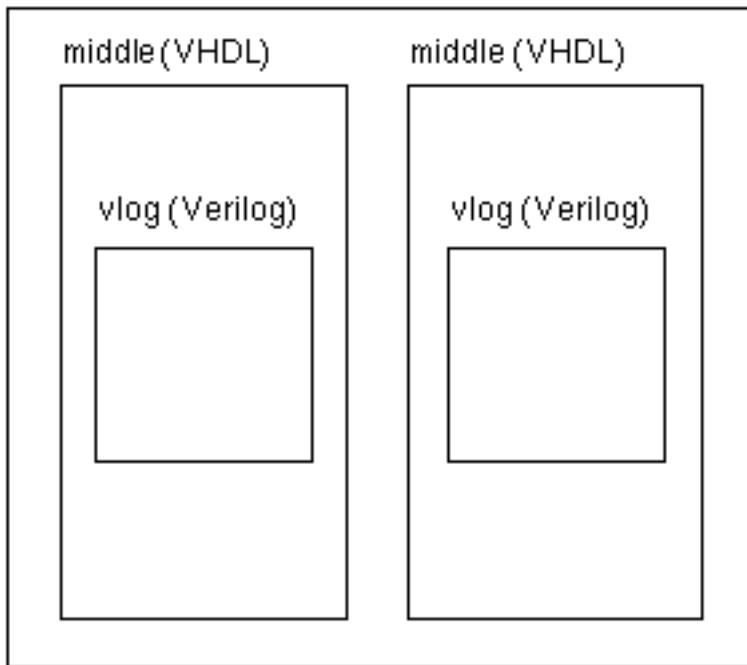
For example, to use *ncverilog* to simulate the design shown in “[Importing VHDL into Verilog without a Shell](#)” on page 375, create the `cds.lib` and `hdl.var` files and then execute the following commands:

```
% ncvhdl foo.vhd  
% ncverilog +mixedlang top.v
```

## A Verilog-VHDL-Verilog Example

In the example shown in this section, a Verilog top-level module, described in the file `top.v`, contains two instantiations of a VHDL block, which is described in the file `middle.vhd`. Each VHDL instantiation has a Verilog child, which is described in the file `sub.v`.

`top(Verilog)`



[“Verilog-VHDL-Verilog Sandwich” on page 1108](#) contains the source code for the example.

You can import the Verilog module into VHDL and then import the VHDL into Verilog with or without a shell.

### Preparing the Design without Shells

To prepare this mixed-language design for simulation without using shells:

1. Compile the Verilog source `sub.v` into the work library.

```
% ncvlog -messages sub.v
```

This command generates `worklib.vlog:module`.

2. Generate a component declaration for the Verilog module. You can write the component declaration manually, or you can use the `ncshell` utility to generate it automatically. See

[“Using ncshell to Generate the Component Declaration”](#) on page 365 for information on using *ncshell* to generate the component declaration.

In this example, the following *ncshell* command is used to generate the component declaration:

```
% ncshell -messages -import verilog -into vhdl worklib.vlog:module
```

This command generates a component declaration in a file called *module\_name\_comp.vhd*. In this example, the component declaration shown below is created in a file called *vlog\_comp.vhd*.

```
library ieee;
use ieee.std_logic_1164.all;

package HDLModels is

component vlog

port (
    io: inout std_logic;
    c0: in std_logic
);
end component;

end HDLModels;
```

3. Cut and paste the component declaration (without the enclosing package statements) into your VHDL code. Then instantiate the component in the VHDL code. You can use either positional or name mapping syntax for the ports.

The following code fragment shows the component declaration and instantiation in the file *middle.vhd*:

```
-- File: middle.vhd
library ieee;
use ieee.std_logic_1164.all;
library worklib;
entity middle is
    port (io : inout std_logic;
          vctrl : in std_logic_vector(1 downto 0));
end middle;
```

```
architecture A of middle is

component vlog
    port (
        io : inout std_logic;
        c0 : in std_logic
    );
end component;

signal ctrl : std_ulogic;

begin
    v1: vlog
        port map(
            io,
            vctrl(1)
        );
    ...
    ...
    ...

```

4. Compile the VHDL source code using *ncvhdl*.

```
% ncvhdl -messages -v93 middle.vhd
```

5. Instantiate the VHDL design unit in your Verilog code using the normal Verilog language mechanism for instantiating components.

You can use either positional or name mapping for the ports.

The following code fragment shows the Verilog instantiations of the VHDL design unit:

```
// File: top.v
module top ;

reg [4:0] vctrl;
reg r_io;
wire c0 = vctrl[4];
wire io = r_io;

middle m10 (io, vctrl[1:0]);
middle m32 (io, vctrl[3:2]);
```

```
always @(io or c0)
...
...
...
...
```

**6. Compile the Verilog code contained in top.v.**

```
% ncvlog -messages top.v
```

This command generates worklib.top:module.

**7. Elaborate the design.**

```
% ncelab -messages worklib.top:module
```

This command generates a simulation snapshot called worklib.top:module, which you can then load into the simulator.

## Preparing the Design with Shells

To prepare this mixed-language design for simulation:

**1. Compile the Verilog source sub.v into the work library.**

```
% ncvlog -messages sub.v
```

This command generates worklib.vlog:module.

**2. Use *ncshell* to generate and compile a VHDL shell for the Verilog model.**

```
% ncshell -messages -import verilog -into vhdl worklib.vlog:module
```

This command generates a VHDL shell called verilog\_module.vhd by default. In this example, the shell file is vlog.vhd.

```
library ieee;
use ieee.std_logic_1164.all;
entity vlog is
    port (
        io: inout std_logic;
        c0: in std_logic
    );
end vlog;
architecture verilog of vlog is
    attribute foreign of verilog:architecture is "VERILOG(event)
                                                worklib.vlog:module";
begin
end;
```

- 3.** Compile the VHDL block contained in the source file called `middle.vhd`.

```
% ncvhdl -messages -v93 middle.vhd
```

This command generates `worklib.middle:a`.

- 4.** Use `ncshell` to generate and compile a Verilog shell for the VHDL model.

```
% ncshell -messages -import vhdl -into verilog worklib.middle:A
```

This command generates a Verilog shell called `middle.vs`. It then updates the `hdl.var` file to include or modify the `VIEW_MAP` variable. The `hdl.var` file now includes the following line:

```
define VIEW_MAP ($VIEW_MAP, .vs => shell)
```

This line specifies that `.vs` files are to be compiled with a view name of `shell`.

`ncshell` then compiles the shell file into `worklib.middle:shell`.

The `middle.vs` shell file generated in this example is as follows:

```
module middle(
    io, vctrl
)
(* const integer foreign = "VHDL(event) WORKLIB.MIDDLE:a"; * );
inout io;
input [1:0] vctrl;
endmodule
```

- 5.** Compile the Verilog file `top.v`.

```
% ncvlog -messages top.v
```

This command generates `worklib.top:module`.

- 6.** Elaborate the design with `ncelab`.

```
% ncelab -messages top
```

This command generates a simulation snapshot called `worklib.top:module`, which you can then load into the simulator.

## Generating a Shell with ncshell

The *ncshell* utility generates a shell file that lets you import Verilog models into VHDL simulations and VHDL models into Verilog simulations.

### ncshell Command Syntax

Invoke *ncshell* with options and arguments. Options can occur in any order. Parameters to options must immediately follow the option they modify. Command-line options can be abbreviated to the shortest unique string, indicated here by capital letters.

```
ncshell -import {vhdl | verilog} -into {vhdl | verilog}  
[other_options] lib.cell:view
```

The *-import* option, which specifies the kind of model (that is, the language of the model) being imported, and the *-into* option, which specifies the kind of model into which import is being done, are required.

The argument is the library.cell:view specification of the compiled design unit that you want to import.

The *ncshell* command-line options listed in this section are divided into the following three groups:

- General options
- Options that you can use when you are importing VHDL into Verilog
- Options that you can use when you are importing Verilog into VHDL

You can use the `NCSHELLOPTS` variable in the `hdl.var` file to specify *ncshell* command-line options.

### General Options

You can use the following options if you are importing Verilog into VHDL or if you are importing VHDL into Verilog.

```
[-ANALyze filename]  
[-ANALopts "compiler_options"]  
[-APPend log]  
[-CDslib filename]  
[-Errormax integer]  
[-FILE filename]
```

```
[-HDLvar filename]  
[-HElp]  
[-IMport {vhdl | verilog}]  
[-INTo {vhdl | verilog}]  
[-LOGfile logfile_name]  
[-Messages]  
[-NEverwarn]  
[-NOCOPyright]  
[-NOCOMpile]  
[-NOLog]  
[-NOSTdout]  
[-NOWarn warning_code]  
[-SHELL shell_output_filename]  
[-VErsion]  
[-Work work_library]
```

## VHDL Models Imported into Verilog

You can use the following options with `-import vhdl -into verilog`.

```
[-Backward]  
[-Generic]  
[-LIst]  
[-SUFfix]  
[-VIew viewname]
```

## Verilog Models Imported into VHDL

You can use the following options with `-import verilog -into vhdl`.

```
[-Backward]  
[-Comp component_output_file]  
[-Generic]  
[-LIst]  
[-NOEscape]  
[-Ulogic]
```

## **ncshell Command Options**

### **-ANALOpt “*compiler\_options*”**

Specifies one or more *ncvhdl* or *ncvlog* command-line options. *ncshell* passes these options to the compiler when it invokes the compiler to compile the HDL source file that you specify with the *-analyze* option.

If you specify more than one option, you must enclose the list in quotation marks. For example:

```
% ncshell -messages -analyze sub.vhd -import vhdl -into verilog  
worklib.sub:A -analopts "-messages -neverwarn -v93"
```

See [Chapter 7, “Compiling Verilog Source Files with ncvlog.”](#) for information on *ncvlog* compiler options. See [“Compiling VHDL Source Files with ncvhdl”](#) in the *NC-VHDL Simulator Help* for information on *ncvhdl* compiler options.

### **-ANALyze *filename***

Specifies the HDL source file that you want to analyze. This option specifies that you want *ncshell* to invoke the compiler on the source file before generating a shell. For example, the following command invokes *ncvhdl* to compile the file *sub.vhd* before invoking *ncshell* to generate the shell.

```
% ncshell -messages -analyze sub.vhd -import vhdl -into verilog worklib.sub:A
```

You do not need to specify this option if the source file has already been analyzed.

Use the *-list* option to analyze multiple files. Use the *-analopts* option to specify command-line options that you want passed to the *ncvlog* or *ncvhdl* compiler.

### **-APPend\_log**

Append log information from multiple runs of *ncshell* to one log file. If you do not use this option, the log file is overwritten each time you run *ncshell*. The *-nolog* option overrides this option.

### **-Backward**

Provides compatibility with Leapfrog VHDL and Verilog-XL shells. NC-Sim model shells use a different syntax than that supported by Leapfrog and Verilog-XL.

Use this option when you want the Leapfrog or Verilog-XL shells to be used in the mixed-language simulation flow instead of the syntax used in NC-Sim shells.

**-CDslib *filename***

Specifies the name of the `cds.lib` file to load. See “[The cds.lib File](#)” on page 110 for more information.

**-CComp *output\_filename***

**(Verilog into VHDL)**

Specifies the filename for the VHDL component declaration.

When you run `ncshell` to generate a VHDL shell to import a Verilog module, `ncshell` also generates a component declaration corresponding to the shell. The default filename is `model_name_comp.vhd` (the model name with `_comp.vhd` appended). Use the `-comp` option to specify a different filename.

**-Errormax *integer***

Abort after reaching the specified number of errors.

**-Ffile *filename***

Use the command-line arguments contained in the specified file.

You can store frequently-used or lengthy command lines by putting command options and arguments in a text file. When you invoke `ncshell` with the `-file` option, the arguments in the specified file are used with the command as if they had been entered on the command line.

You must specify each option and its arguments on a separate line.

**-Generic**

**(Verilog or VHDL model import)**

Convert the VHDL generic data types to Verilog parameter declarations, or convert Verilog parameter declarations to VHDL generic data types.

By default, *ncshell* creates only the port interface in the shell. The VHDL generic declarations and Verilog parameter declarations are not included. Use the `-generic` option when you want VHDL generic data types converted to Verilog parameter declarations or when you want Verilog parameter declarations converted to VHDL generics.

Verilog requires parameters to have default values. The *ncshell* utility assigns the default value `1'bX` to any VHDL generic data type that does not have a default value.

**-HDIvar *filename***

Specifies the `hdl.var` file to load. See “[The hdl.var File](#)” on page 118 for information on the `hdl.var` file.

**-HEIp**

Display a list of the `ncshell` command options.

**-IMport {verilog | vhdl}**

Specifies the type of model that you are importing. This option is required for every model import.

**-INto {vhdl | verilog}**

Specifies the language into which you are importing a design unit. Use the `vhdl` argument when you want to import a Verilog module into VHDL. Use the `verilog` argument when you want to import a VHDL design unit into Verilog. This option is required for every model import.

**-Llist *list\_filename***

Specifies a file containing a work library name and a list of files that you want to analyze. Use this option to analyze multiple files at one time.

A line in the file *list\_filename* has the following syntax:

```
<work_library_name> file_name file_name ...
```

Enclose the work library within angle brackets, and separate file names with spaces.

For example, the following line analyzes the files named `alu`, `shift` and `control` and places the generated views into the library named `worklib`.

```
<worklib> alu shift control
```

You do not need to analyze a file if it has been analyzed once before. Use the `-analyze` option to analyze a single source file.

#### **-LOgfile *logfile\_name***

Use the specified name for the log file instead of the default name `ncshell.log`.

Use `-nolog` if you don't want a log file. If you use both `-logfile` and `-nolog` on the command line, `-logfile` overrides `-nolog`.

#### **-Messages**

Display informational messages during the creation of the shell.

**Note:** Information is written to the log file only when you use the `-messages` option.

#### **-NEverwarn**

Disable printing of all warning messages.

#### **-NOCOPright**

Suppress the display of the copyright banner.

#### **-NOCOMpile**

Do not compile the shell.

By default, `ncshell` automatically compiles the shell that it generates. Use this option if you do not want `ncshell` to compile the shell.

### **-NOEscape**

#### **(Verilog into VHDL)**

Do not escape names in the VHDL shell.

By default, *ncshell* escapes mixed-case and uppercase identifiers in the VHDL shell. For example, if the Verilog module is `vlog`, this identifier appears in the VHDL shell as `\vlog\`. Use the `-noescape` option if you want the Verilog module name to be matched exactly in the shell.

Do not set the `CDS_ALT_NMP` environment variable. This variable is not supported.

### **-NOLog**

Do not generate a log file. By default, *ncshell* generates a log file called `ncshell.log`.

### **-NOStdout**

Suppress printing of output to the screen.

### **-NOWarn *warning\_code***

Disable printing of the warning with the specified code. The *warning\_code* argument is the message code (mnemonic) that appears in the warning message following the error severity code.

### **-SHell *shellname***

Use the specified name for the shell.

By default, the shell name is the same as the primary design unit or module name with a `.vhd` (for VHDL) or `.vs` (for Verilog) extension. Use the `-shell` option to give the generated shell a name other than the default name.

**-Suffix *file\_extension***

**(VHDL into Verilog)**

Specifies a filename extension for a Verilog shell. This option is ignored if you also specify the `-shell` option.

**-Ulogic**

**(Verilog into VHDL)**

Generate a shell with `std_ulogic` ports.

By default, `ncshell` generates a VHDL shell in which the port type is `std_logic`. Use this option if you want `ncshell` to generate a shell in which all ports are of type `std_ulogic`.

This option can only be used with `-import verilog -into vhdl`.

**-View *viewname***

**(VHDL into Verilog)**

Specifies the name of the view that you want the generated shell analyzed into. The default view name is `shell`.

**-VErsion**

Display the version of `ncshell` and exit.

**-Work *work\_library***

Use the specified library as the work library. The `ncshell` utility stores analyzed models in the default work library, which is defined by the `WORK` variable in your `hdl.var` file. Use this option to store analyzed models in a library other than the default.

## Importing a Verilog-XL Design into VHDL

In Verilog-XL, you can specify how you want the simulator to search libraries for instance definitions by using the `-y` and `-v` command-line options and with the `'uselib` compiler directive. These constructs often create a different search order than the default search order used by `ncelab`, particularly in cases where there is more than one description of the design.

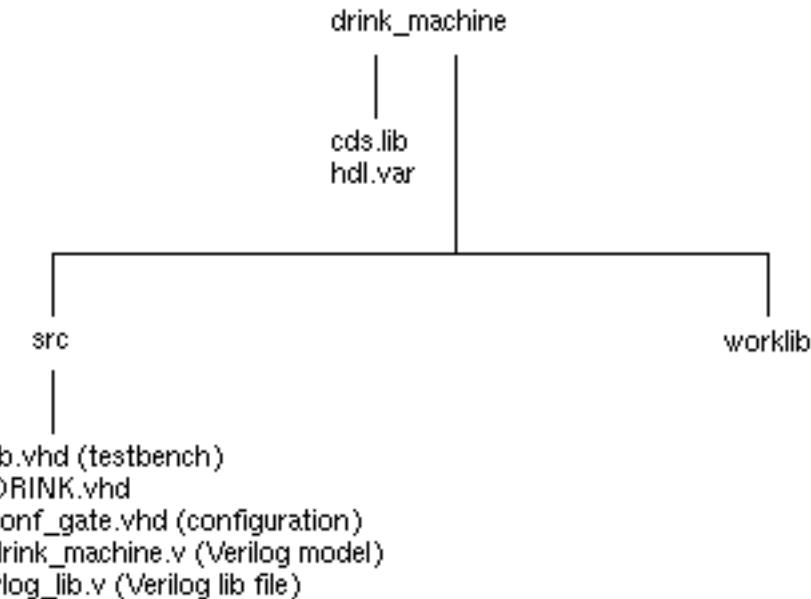
Another problem can arise with macros that you define using `'define`. In Verilog designs, `'define` statements are often used in one Verilog file to create a macro definition that is used in other Verilog files. When the `ncvlog` compiler compiles these other Verilog files, the macro is not defined.

In these situations, you need to use `ncverilog` with the `+import` option to compile your Verilog design instead of using `ncvlog`. `ncverilog` uses the same library search order that Verilog-XL uses, duplicates the binding rules of XL, and propagates macros in the same way that XL does.

See [Chapter 4, “Running NC-Verilog with the ncverilog Command,”](#) for full details on `ncverilog`. This section discusses `ncverilog` only as it is used to prepare your Verilog design for import.

The following examples present two variations on using `ncverilog` to compile a Verilog-XL design. In the first example, `ncverilog` creates libraries in the `INCA_libs` directory. This is the default for `ncverilog`. The second example shows you how to control the location of the Verilog libraries by modifying the `cds.lib` and `hdl.var` files so that `ncverilog` does not create new libraries in an `INCA_libs` directory.

Both examples use a VHDL RTL design of a vending machine that dispenses soft drinks. The design contains a Verilog block modeled at the gate level. The directory structure is as follows:



All of the source files for this design are in the `src` directory. The `worklib` directory is the physical location of a library that has the logical name `worklib`.

### Example 1

In this example, you use `ncverilog` to compile a Verilog-XL design for import into VHDL. `ncverilog` creates an `INCA_libs` directory to contain the Verilog design library specified on the command line.

1. Define libraries for the compiled VHDL objects in the `cds.lib` file. The `cds.lib` file for this example is as follows:

```
# cds.lib
INCLUDE your_install_directory/tools/inca/files/cds.lib
DEFINE worklib ./worklib
```

2. Define the work library in the `hdl.var` file. The `hdl.var` file for this example is as follows:

```
# hdl.var
INCLUDE your_install_directory/tools/inca/files/hdl.var
DEFINE WORK worklib
```

3. Compile the Verilog model using `ncverilog +import`. To do this, replace the `verilog` command on the Verilog-XL command line with `ncverilog +import`. It is recommended that you also include the `+crshell` option to automatically invoke `ncshell` to generate a VHDL shell and to compile it.

```
% ncverilog +import +crshell -v src/vlog_lib.v +libext+.v src/drink_machine.v
```

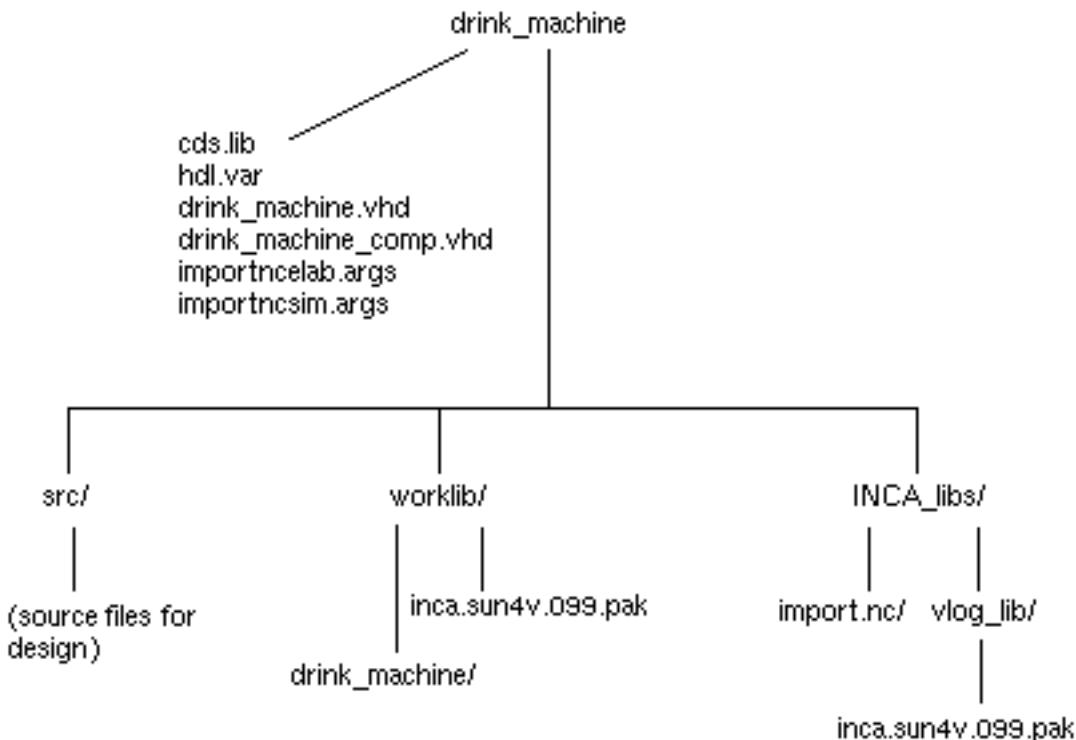
This command invokes `ncverilog`, which:

- ❑ Translates the Verilog-XL command-line options to the corresponding options for `ncvlog`, `ncelab`, and `ncsim`. `ncverilog` writes the arguments for `ncelab` to a file called `importncelab.args` and writes the arguments for `ncsim` to a file called `importncsim.args` by default. Use `+ncelabfile` and `+ncsimfile` to specify a different name for these files. Use the `-file` option to include these argument files when you invoke the elaborator and simulator.
- ❑ Compiles the Verilog model. The `drink_machine` module (described in the source file `src/drink_machine.v`) is compiled into the work library `worklib`.

By default, `ncverilog` creates a directory called `INCA_libs` to contain new libraries. The `INCA_libs` directory contains directories for each `-v` library file and for each `-y` library directory. The names of the directories are the names of the library files and directories. In this example, the `INCA_libs` directory contains a subdirectory called `vlog_lib`.

- ❑ Updates the `cds.lib` and `hdl.var` files to reflect the new libraries.
- ❑ Invokes the `ncshell` utility to generate a VHDL shell for the Verilog model. The shell is then automatically compiled into the work library (`worklib`). The shell is called `verilog_module_name.vhd`. In this example, `ncshell` generates a shell called `drink_machine.vhd`, which `ncverilog` then compiles. The `ncshell` utility also generates a component declaration called `verilog_module_name_comp.vhd` by default. In this example, the component declaration file is called `drink_machine_comp.vhd`.

After running *ncverilog*, the directory structure is as follows:



4. Edit the VHDL configuration so that the model import shell is used. By default, the entity name of the shell is the same as the top-level Verilog module, and the architecture name is *verilog*.
5. Compile the VHDL code with *ncvhdl*:
 

```
% ncvhdl src/DRINK.vhd
% ncvhdl drink_machine.vhd
% ncvhdl src/tb.vhd
% ncvhdl src/conf_gate.vhd
```
6. Elaborate the design with *ncelab*. Use the *-file* option to include the *importncelab.args* file:
 

```
% ncelab -file importncelab.args worklib.cfg_gate
```
7. Invoke the simulator. Use the *-file* option to include the *importncsim.args* file:
 

```
% ncsim -file importncsim.args worklib.cfg_gate
```

## Example 2

This example uses the same design example to illustrate how to use *ncverilog* to compile a Verilog-XL design for import into VHDL, but in this example, you control the location of your design libraries.

1. Define libraries for both the compiled VHDL and Verilog objects in the `cds.lib` file. The `cds.lib` file for this example is as follows:

```
# cds.lib
INCLUDE your_install_directory/tools/inca/files/cds.lib
DEFINE worklib ./worklib
DEFINE vlog_lib ./vlog_lib
```

2. Define the work library in the `hdl.var` file. Then define the `LIB_MAP` variable to specify the library (or libraries) into which modules are to be compiled. The `hdl.var` file for this example is as follows:

```
# hdl.var
DEFINE WORK worklib
DEFINE LIB_MAP (./src/vlog_lib.v => vlog_lib)
```

This `LIB_MAP` variable tells *ncvlog* to compile `src/vlog_lib.v` into the library called `vlog_lib`. See “[LIB\\_MAP](#)” on page 121 for details on the `LIB_MAP` variable.

3. Compile the Verilog model using `ncverilog +import`. Use the `+crshell` option to automatically invoke *ncshell* to generate a VHDL shell and to compile it:

```
% ncverilog +import +crshell -v src/vlog_lib.v +libext+.v src/drink_machine.v
```

4. Edit the VHDL configuration so that the model import shell is used.

5. Compile the VHDL code, including the shell file, with *ncvhdl*.

6. Elaborate the design with *ncelab*. Use the `-file` option to include the `importncelab.args` file.

7. Invoke the simulator. Use the `-file` option to include the `importncsim.args` file.

Here is another example of how you can modify the `cds.lib` and `hdl.var` files to control the location of your Verilog libraries. Suppose that your Verilog-XL model has two library directories called `vlogsrc` and `CELL`. The command line that you used to invoke Verilog-XL was:

```
% verilog -y ../vlogsrc -y ../CELL +libext+.v+.vg top.v
```

You can create a `cds.lib` file that defines three libraries, as follows:

```
# cds.lib
DEFINE worklib ./worklib
DEFINE srclib ./srclib
DEFINE celllib ./celllib
```

Then create an `hdl.var` file that defines the work library and that uses the `LIB_MAP` variable to tell `ncverilog` where to compile the Verilog code. For example, you might create the following `hdl.var` file:

```
# hdl.var
DEFINE WORK worklib
DEFINE LIB_MAP (../vlogsrc => srclib, \
                ../CELL => celllib, \
                + => worklib)
```

When you run `ncverilog +import`:

- All files in `../vlogsrc` are compiled into the library called `srclib` (`./srclib`).
- All files in `../CELL` are compiled into the library called `celllib` (`./celllib`).
- All other files are compiled into the library called `worklib` (`./worklib`).

## Preparing a Leapfrog Verilog Model Import Design

If you are a Leapfrog Verilog Model Import customer, you already have:

- A `cds.lib` file to define your libraries.
- An `hdl.var` file that defines the work library and that defines the `VXLOPTS` variable to specify command-line arguments for Verilog-XL.
- A model import shell for importing the Verilog model.

Leapfrog-style shells contain a foreign attribute string that indicates the name of the top-level Verilog module and the filename that contains this top-level module. The following is an example foreign attribute string:

```
attribute foreign of verilog:architecture is  
    "VERILOG:drink_machine src/drink_machine.v";
```

Cadence provides a utility called *ncximport* to help you prepare your Verilog design for import into NC-Sim. This utility reads the Leapfrog-style shell and the `VXLOPTS` variable and then invokes *ncverilog* with the command-line arguments specified with the variable. *ncverilog* does not generate a new shell.

This section describes the *ncximport* utility and then shows you how to use it to compile your Verilog source. The example uses the same drink vending machine example used in the previous sections.

### The *ncximport* Utility

The *ncximport* utility reads a Leapfrog-style shell and the `VXLOPTS` variable in the `hdl.var` file. It then invokes *ncverilog* with the command-line arguments specified with the `VXLOPTS` variable and compiles the Verilog source that you want to import into a VHDL model.

By default, *ncverilog* creates a directory called `INCA_libs` to contain any libraries. You can control the location of your design libraries by editing the `cds.lib` file and by using the `LIB_MAP` variable in the `hdl.var` file. This methodology is described in “[Example 2](#)” on page 397.

## **ncxlimport Syntax**

The syntax of the `ncxlimport` command is as follows:

```
ncxlimport [options] [lib.]cell[:view]
```

The `[lib.]cell[:view]` argument is the library, cell, and view of the compiled (with *ncvhdl*) Leapfrog-style model import shell.

If you do not specify the library, the default is the work library that you specified with the `-work` option or with the `WORK` variable in the `hdl.var` file. The `-work` option overrides the definition of the `WORK` variable.

If you do not specify the view, the default is `verilog`, which is the default architecture for Leapfrog-style model import shells created using the `verilog +vhdl_crshell filename` command.

## **ncxlimport Command Options**

### **-Append\_log**

Append log information from multiple runs of *ncxlimport* to one log file. If you do not use this option, the log file is overwritten each time you run *ncxlimport*.

### **-Errormax integer**

Abort after reaching the specified number of errors.

### **-Help**

Display a list of the `ncxlimport` command-line options with a brief description of each option.

### **-LogFile *filename***

Use the specified name for the log file instead of the default name `ncxlimport.log`.

### **-Messages**

Display informative messages during execution.

### **-NCELabfile *filename***

Use the specified name for the file that contains arguments to *ncelab*, instead of using the default name `importncelab.args`. Use the `-file` option to include this file when you elaborate the design.

### **-NCERror *warning\_code***

Increase the severity level of the specified warning message from warning to error. The *warning\_code* argument is the message code (mnemonic) that appears in the warning message following the severity code.

Example:

```
% ncxlimport -messages -ncerror ABCDEF worklib.drink_machine:verilog
```

You can include multiple `-ncerror` options on the command line.

Using this option can change the behavior of the tool because functions that return errors instead of warnings may behave differently.

### **-NCFatal {*warning\_code* | *error\_code*}**

Increase the severity level of the specified warning message or error message from warning or error to fatal. The *warning\_code* or *error\_code* argument is the message code (mnemonic) that appears in the message following the severity code.

Example:

```
% ncxlimport -messages -ncfatal LMNOPQ worklib.drink_machine:verilog
```

You can include multiple `-ncfatal` options on the command line.

### **-NCSimfile *filename***

Use the specified name for the file that contains arguments to *ncsim*, instead of using the default name `importncsim.args`. Use the `-file` option to include this file when you invoke the *ncsim* simulator.

### **-NEverwarn**

Disable the display of all warning messages.

**-NOCopyright**

Suppress the display of the copyright banner.

**-NOLog**

Do not generate a log file. By default, *ncximport* generates a log file called `ncximport.log`.

If you use both `-nolog` and `-logfile` on the command line, `-logfile` overrides `-nolog`.

**-NOStdout**

Suppress the display of output to the screen.

**-NOVxopts**

Ignore the `VXLOPTS` variable in the `hdl.var` file.

**-NOWarn *warning\_code***

Disable the display of the warning that has the specified code. The *warning\_code* argument is the message code (mnemonic) that appears in the warning message following the error severity code.

**-Version**

Display the version of *ncximport* and exit.

**-Work *work\_library\_name***

Use the specified library as the *work* library.

**Example:**

In this example, you use *ncximport* to compile a Verilog-XL design for import into VHDL. The *ncximport* utility invokes *ncverilog*, which creates an *INCA\_libs* directory to contain the Verilog design library specified with the *VXLOPTS* variable in the *hdl.var* file.

This example uses the drink vending machine example used in previous sections.

The *cds.lib* file defines a library called *worklib*, as follows:

```
# cds.lib
INCLUDE your_install_directory/tools/inca/files/cds.lib
DEFINE worklib ./worklib
```

The *hdl.var* file defines the work library as *worklib* and includes a definition of the *VXLOPTS* variable as follows:

```
# hdl.var file
DEFINE WORK worklib
DEFINE VXLOPTS -v ./src/vlog_lib.v +maxdelays +max_err_count+25 -s
```

The shell file for the Verilog block is called *drink\_machine\_vlog\_imp.vhd*. This is a Leapfrog-style shell, as shown below:

```
library ieee;
use ieee.std_logic_1164.all;
entity drink_machine is
    port (
        nickel_in: in std_logic;
        dime_in: in std_logic;
        quarter_in: in std_logic;
        reset: in std_logic;
        clk: in std_logic;
        nickel_out: out std_logic;
        dime_out: out std_logic;
        two_dime_out: out std_logic;
        dispense: out std_logic
    );
end drink_machine;
architecture verilog of drink_machine is
    attribute foreign of verilog:architecture is
        "VERILOG:drink_machine src/drink_machine.v";
begin
end;
```

To prepare this design for simulation with NC-Sim:

1. Compile all of the VHDL code, including the model import shell:

```
% ncvhdl -messages src/drink_machine_vlog_imp.vhd  
% ncvhdl -messages src/DRINK.vhd  
% ncvhdl -messages src/tb.vhd  
% ncvhdl -messages src/conf_gate.vhd
```

The shell is compiled into the work library (`worklib`) with a default architecture name of `verilog`.

2. Run the `ncximport` utility. The argument to the `ncximport` command is the `lib.cell:view` specification for the compiled shell.

```
% ncximport -messages worklib.drink_machine:verilog
```

`ncximport` reads the shell and the `VXLOPTS` variable and then invokes `ncverilog` to compile the Verilog model. `ncverilog` translates the Verilog-XL command-line options to corresponding options for `ncvlog`, `ncelab`, and `ncsim`. Arguments to `ncelab` are written to a file called `importncelab.args` by default. Arguments to `ncsim` are written to a file called `importncsim.args` by default. Use the `-ncelabfile` and `-ncsimfile` options to specify different names for these files.

By default, `ncverilog` creates a directory called `INCA_libs` to contain new libraries. The `INCA_libs` directory contains directories for each `-v` library file and for each `-y` library directory. The directory structure that `ncverilog` creates with this example is the same as that shown in “[Example 1](#)” on page 394. As in that example, `ncverilog` modifies the `cds.lib` file to reflect any new libraries, and it modifies the `hdl.var` file to include definitions of the `LIB_MAP` and `VIEW_MAP` variables. After running `ncximport` on this example, these files are as follows:

```
#cds.lib  
INCLUDE your_install_directory/tools/inca/files/cds.lib  
DEFINE worklib ./worklib  
DEFINE vlog_lib ./INCA_libs/vlog_lib  
  
# hdl.var  
DEFINE WORK worklib  
DEFINE VXLOPTS -v ./src/vlog_lib.v +maxdelays +max_err_count+25 -s  
DEFINE VIEW_MAP ( $VIEW_MAP, .v => v)  
DEFINE LIB_MAP ( $LIB_MAP, ./src/vlog_lib.v => vlog_lib )
```

3. Elaborate the design with `ncelab`. Use the `-file` option to include `importncelab.args`:

```
% ncelab -messages -file importncelab.args worklib.cfg_gate
```

4. Invoke the simulator. Use the `-file` option to include `importncsim.args`:

```
% ncsim -messages -file importncsim.args worklib.cfg_gate
```

## Mixed-Language Out-of-Module References

This section describes how to write hierarchical path references in your Verilog models that refer to objects in other Verilog islands that go through VHDL hierarchies. These references are called *out-of-module references* (OOMR).

The first component in an OOMR can be either VHDL or Verilog. This can be followed by any sequence of VHDL and Verilog components. The path must end with a name that refers to a Verilog object or scope.

```
vhdl_entity_name.....any_sequence_of_components.....verilog_object	scope  
verilog_module_name....any_sequence_of_components.....verilog_object	scope
```

**Note:** No hierarchical path can end with a name that refers to a VHDL object or scope. In other words, out-of-architecture references are not allowed from within a Verilog module.

If the VHDL name is in lowercase, non-escaped, and is not a Verilog keyword, you can use the same name in the OOMR. For example, if the VHDL name is `first_inst`, you can use `first_inst` in the OOMR.

If the VHDL name is in uppercase, mixed-case, escaped, or a Verilog keyword, use the *nm*p utility to determine the correct VHDL to Verilog name mapping for a VHDL instance or scope name that forms part of the hierarchical path to the Verilog object. Then use the name-mapped Verilog object name as part of the hierarchical path in the OOMR references. For example, if the VHDL name is `First_Inst`, you must use the *nm*p utility to determine the name to use in the OOMR.

Invoke the *nm*p utility as follows:

```
% nm mapName VHDL Verilog vhdl_name
```

For example,

```
% nm mapName VHDL Verilog First_Inst
```

Only integers are allowed as part of index specifications along with for-generate labels. That is, if `my_instances` is a for-generate label that runs from 1 to 5, only references of the form `my_instances(3)` or `my_instances(5)` are valid VHDL names that can be name mapped and used in the OOMR references. The name mapping of array instances, such as `iter(1)` is straightforward: Change the parentheses to square brackets (`iter[1]`, for example).

## Example

Consider the following Verilog module:

```
module bot(i, o);
    input i;
    output o;
    reg o;

    initial
        begin
            assign o = ~i;
        end

    initial
        $monitor("in1: %b, out1: %b, in2: %b, out2: %b",
            top.iter[1].first_inst.inst1.i,
            top.iter[1].first_inst.inst1.o,
            top.iter[2].second_inst.inst2.i,
            top.iter[2].second_inst.inst2.o);
endmodule
```

A VHDL design that instantiates the Verilog model `bot` and that also applies stimulus, is shown below:

```
library ieee;
use ieee.std_logic_1164.all;

entity top is
end top;

architecture a of top is
    signal s_in1, s_in2, s_out1, s_out2: std_logic;
    component bot is
        port (i: in std_logic;
              o: out std_logic);
    end component bot;

begin
    iter: for i in 1 to 2 generate
        begin
            First_Inst: if i = 1 generate
                begin
                    INST1: bot
                        port map (s_in1, s_out1);
                end generate First_Inst;
            end if;
        end generate iter;
end architecture;
```

```
Second_Inst: if i /= 1 generate
begin
    INST2: bot
        port map (s_in2, s_out2);
    end generate Second_Inst;
end generate iter;

test_process: process
begin
    s_in1 <= '1';
    s_in2 <= '0';
    wait for 10 ns;
    s_in1 <= '0';
    s_in2 <= '1';
    wait;
end process;

end;
```

To determine how to write the hierarchical OOMR references used in the `$monitor` statement in the Verilog module, run the *nmp* utility. For example, one of the OOMR references used in the `$monitor` statement is:

```
top.iter[1].first_inst.inst1.i
```

Run the *nmp* utility as follows to determine the OOMR hierarchical path to be written in the Verilog source.

```
% nmp mapName VHDL Verilog top
top
```

(The VHDL `iter(1)` becomes `iter[1]`

```
% nmp mapName VHDL Verilog First_Inst
first_inst
% nmp mapName VHDL Verilog INST1
inst1
```

The final object referred to in the path (object `i`) is an object in a Verilog scope, so it is not necessary to map this name.

## Path Names and Mixed-Language Designs

In Verilog, you use a period to separate path elements, and paths never begin with a path element separator. If the first element of the path is an item in the debug scope, the simulator assumes that the name is relative. If not, it is assumed to be full, and the first element must be the name of a top-level module.

The following is an example of a Verilog path:

```
board.counter.a
```

In VHDL, you use a colon to separate path elements. A full path begins with a colon, which represents the top-level design unit. The first path element is an item in the top-level scope. The following are examples of fully specified paths:

```
:vending  
:vending:drinks  
:vending:drinks:sig2
```

Relative paths do not begin with a colon. For example, if the current debug scope is :vending, the path name drinks refers to a scope within the scope vending, which is within the top-level design unit.

In a mixed-language simulation, you can use a period or a colon as the path element separator. NC-Sim uses the following rules:

- If the path begins with a colon, the path is a full path name starting at the VHDL top-level scope. A colon by itself refers to this scope. You cannot use any other special character at the start of a path.
- If the path does not start with a colon, and the first path element is in the current debug scope, the path is relative to the debug scope. If the first path element is not in the current debug scope, the simulator assumes that the path is a full path name whose first path element is the name of one of the top-level Verilog modules.

For example, suppose that you have a mixed Verilog-VHDL design, where the top-level design unit is VHDL. With Tcl commands, you can use both path element separators

interchangeably (except at the beginning of a path, as specified above), as shown in the following examples:

```
ncsim> scope -set :board:counter:a
ncsim> scope -set board:counter.a
ncsim> scope -set board.counter:a
```

Because VHDL is case insensitive (except for escaped names) and Verilog is case sensitive, each element of a mixed-language path is either case sensitive or case insensitive, depending on its language context. When the parser looks for a name in a Verilog scope, it is case sensitive; when it looks for a name in a VHDL scope, it is case insensitive.

The syntax that you use for name expressions is also interchangeable. Name expressions are bit-selects, part-selects, and array element specifiers in Verilog, and array element and record field specifiers in VHDL. Index specifiers are also used in VHDL scope names when the scope is created by a for-generate statement.

Verilog index specifiers use square brackets, and a colon separates the left and right bounds of the range (for example [ 7 : 0 ]). VHDL index specifiers use parentheses, and the keyword TO or DOWNTO separates the left and right bounds of the range (for example, ( 7 downto 0 )).

You can use either style with VHDL index ranges. Using a colon in a VHDL index range is the same as using the direction with which that index range was declared.

Record field specifiers apply only to VHDL objects. Use a period to separate the object name from the record field.

The following pairs of Tcl commands are identical.

```
ncsim> scope foo_array(2)
ncsim> scope foo_array[2]

ncsim> value sig[7:0]
ncsim> value sig(7:0)

ncsim> value sig[7]
ncsim> value sig(7)

ncsim> describe sig[7 downto 0]
ncsim> describe "sig(7 downto 0)"
```

You can use either Verilog or VHDL escaped name syntax in path names. For Verilog, escaped names begin with a backslash and are terminated with white space. For example, notice the white space after `\some_name` in the following example:

```
abc.xyz.\some_name .signal
```

For VHDL, escaped names begin and end with a backslash (for example, `\w3.OUT\`).

The following two `value` commands are identical:

```
ncsim> value top.vending.@{\w3.OUT }  
ncsim> value top.vending.@{\w3.OUT\}
```

## SDF Annotation for Mixed-Language Designs

You can annotate the timing check and delay data in an SDF file to Verilog and to VHDL VITAL. See [Chapter 17, “SDF Timing Annotation.”](#) for information on SDF annotation.

## Generating a Value Change Dump (VCD) File for a Mixed-Language Design

A value change dump (VCD) file is an ASCII file that contains information about value changes on selected variables in the design. The file contains header information, variable definitions, and the value changes for all specified variables.

See the *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language* (IEEE Std 1364-1995 or 1364-2001) for details on the syntax and format of the VCD file.

**Note:** You can only dump objects that have read access. If you specify a scope as an argument to the `probe` command, objects within that scope that do not have read access are excluded from the dump, and the simulator prints a warning message. If you specify an individual variable and that object does not have read access, the simulator prints an error message. See [“Enabling Read, Write, or Connectivity Access to Simulation Objects”](#) on page 248 for details on specifying access to simulation objects.

To generate a VCD file for a mixed Verilog/VHDL design:

1. Open a VCD database with the **Tcl** database **-open -vcd** command. The syntax is as follows:

```
database [-open] dbase_name -vcd  
[-default]  
[-into filename]  
[-maxsize max_byte_size]  
[-timescale timescale_value]
```

The following command opens a default VCD database named `vcddb`. The filename is `sim.dump`. The `-timescale` option sets the `$timescale` value in the VCD file to 1 ns. Value changes in the VCD file are scaled to 1 ns.

```
ncsim> database -open vcddb -vcd -default -into sim.dump -timescale ns  
Created VCD database vcddb
```

See “[database](#)” on page 517 for details on the `database` command.

2. Probe signals to the database with the **probe -create -vcd** command. The syntax is as follows:

```
probe [-create] [{object | scope_name}... ] {-vcd | -database dbase_name}  
[-all]  
[-depth {n | all | to_cells}]  
[-inputs]  
[-name probe_name]  
[-outputs]  
[-ports]  
[-screen [-format format_string] [-redirect filename] objects]  
[-variables]
```

The optional `-create` modifier can be followed by an argument that specifies:

- The object(s) to be traced
- The scope(s) to be traced
- A combination of object(s) and scope(s) to be traced

If you do not specify an argument, the current debug scope is assumed, but you must include an option that specifies which objects to include in the trace (`-all`, `-inputs`, `-outputs`, or `-ports`).

If more than one database is open, you must include an option to specify the database into which you want to dump values. You can do this either by specifying a database name with the `-database` option or by using the `-vcd` option to send the probe to the default VCD database. If no default database is open, the simulator opens a default database called `ncsim.vcd`.

The following `probe` command creates a probe on all ports in the scope `top.counter`. Data is sent to the default VCD database.

```
ncsim> probe -create -vcd top.counter -ports  
Created probe 1
```

See “[probe](#)” on page 571 for details on the `probe` command.

## Example

In the following example, the design shown in “[A Verilog-VHDL-Verilog Example](#)” on page 380 is used to illustrate how to generate a VCD file for a mixed-language design. “[Verilog-VHDL-Verilog Sandwich](#)” on page 1108 contains the source code for the example.

1. Compile the Verilog and VHDL source files with the following commands:

```
% ncvlog sub.v  
% ncvhdl -v93 middle.vhd  
% ncvlog top.v
```

2. Elaborate the design with the following command. The `-access` option provides read access to simulation objects.

```
% ncelab -access +r worklib.top:module
```

The elaborator generates a snapshot called `worklib.top:module`.

3. Load the snapshot into the simulator with the following command:

```
% ncsim -tcl worklib.top:module
```

4. Open a VCD database. The following command opens a default VCD database named `vcddb`. The filename is `sim.dump`.

```
ncsim> database -open vcddb -vcd -default -into sim.dump  
Created VCD database vcddb
```

5. Probe signals to the database. The following command probes all signals in the design to the default VCD database.

```
ncsim> probe -create -vcd -all -depth all  
Created probe 1
```

6. Run the simulation.

The following shows the output file, `sim.dump`.

```
$date  
Jul 3, 2001 15:52:22  
$end  
$version  
ncsim: v03.30.(p001)
```

## NC-Verilog Simulator Help

### Mixed Verilog/VHDL Simulation

---

```
$end
$timescale
  1 fs
$end

$scope module top $end
$var reg      5 !    vctrl [4:0] $end
$var reg      1 "    r_io   $end
$var wire     1 #    c0   $end
$var wire     1 $    io   $end

$scope module m10 $end
$var wire     1 %    ctrl   $end
$var wire     1 &    io   $end
$var wire     1 '    vctrl [1] $end
$var wire     1 (    vctrl [0] $end

$scope module v1 $end
$var wire     1 )    io   $end
$var wire     1 *    c0   $end
$var reg      1 +    r_io   $end
$upscope $end

$upscope $end

$scope module m32 $end
$var wire     1 ,    ctrl   $end
$var wire     1 &    io   $end
$var wire     1 -    vctrl [1] $end
$var wire     1 .    vctrl [0] $end

$scope module v1 $end
$var wire     1 /    io   $end
$var wire     1 0    c0   $end
$var reg      1 1    r_io   $end
$upscope $end

$upscope $end

$upscope $end

$enddefinitions $end
$dumpvars
b0 !
```

## NC-Verilog Simulator Help

### Mixed Verilog/VHDL Simulation

---

```
z "
0#
z$ 
0%
Z&
0'
0(
z )
0*
z+
0,
0-
0.
z/
00
z1
$end
#200000000
b1 !
1(
1%
x$ 
X&
x/
x)
#220000000
0$ 
0&
0/
0)
#230000000
1$ 
1&
1/
1)
#240000000
...
...
...
#1880000000
x$
```

## **NC-Verilog Simulator Help**

### Mixed Verilog/VHDL Simulation

---

```
X&
x/
x)
#1890000000
z$ 
Z&
z /
z )
#1890000000
```

## Debugging Your Design

---

This chapter contains the following sections:

- [Managing Databases](#)
- [Setting and Deleting Probes](#)
- [Traversing the Model Hierarchy](#)
- [Setting Breakpoints](#)
- [Disabling, Enabling, Deleting, and Displaying Breakpoints](#)
- [Stepping Through Lines of Code](#)
- [Forcing and Releasing Signal Values](#)
- [Depositing Values to Signals](#)
- [Displaying Information About Simulation Objects](#)
- [Displaying the Drivers of Signals](#)
- [Checking for Bus Contention and Bus Float Conditions](#)
- [Displaying Waveforms with SimVision Waveform Viewer](#)
- [Generating a Value Change Dump \(VCD\) File](#)
- [Generating an Extended Value Change Dump \(EVCD\) File](#)
- [Comparing Databases with Comparescan](#)
- [Generating a Code Coverage Database File](#)
- [Displaying Debug Settings](#)
- [Setting a Default Radix](#)
- [Setting Variables](#)
- [Suppressing Assert Messages in IEEE or User-Defined Packages](#)

## **NC-Verilog Simulator Help**

### Debugging Your Design

---

- [Editing a Source File](#)
- [Searching for a Line Number in the Source Code](#)
- [Searching for a Text String in the Source Code](#)
- [Configuring Your Simulation Environment](#)
- [Saving and Restoring Your Simulation Environment](#)
- [Creating or Deleting an Alias](#)
- [Getting a History of Commands](#)
- [Managing Custom Buttons](#)

## Managing Databases

You can open, close, disable, enable, and display information about databases.

### Opening a Database

You can open three types of databases:

- SHM for Verilog, VHDL, or mixed-language
- Value Change Dump (VCD) for Verilog, VHDL, or mixed-language
- Extended Value Change Dump (EVCD) for Verilog or VHDL

If you are using the Tcl command-line interface, use the `database` command with the optional `-open` modifier to open a database. You must specify a database name. There are three command-line options that you can use to specify the type of database that you want to open: `-shm`, `-vcd`, and `-evcd`. By default, `ncsim` opens an SHM database.

**Note:** For Verilog, you cannot use the `database -open -evcd` command to open an EVCD database. Use the `$dumpports` system task in your Verilog code to open an EVCD database.

The basic syntax of the `database` command is as follows:

```
database [-open] dbase_name [ {-shm | -vcd | -evcd} ]
```

See “[database](#)” on page 517 for details on the `database` command.

If you are using the SimVision analysis environment, select *File–SHM Database–Open* and fill in the Open Database form to open an SHM database. To open a VCD database or an EVCD database, you must use the `database` text command. Enter the `database` command at the prompt in the I/O region of the SimControl window.

See “[Managing Databases](#)” in the *SimVision Analysis Environment User Guide* for an example of using the commands on the SimControl window for managing your databases.

For Verilog, you can also open an SHM database with the `$shm_open` system task in your Verilog code. The name of the database that is created is preceded by an underscore character. For example, the following system task opens a database called `_waves.shm`.

```
$shm_open( "waves.shm" );
```

This lets you interact with databases opened with `$shm_open` in the same way that you interact with databases that you open with the `database` command.

For VHDL, you can also open a VCD database and probe objects to the database by using the `call` command to call predefined CFC routines, which are part of the NC VHDL simulator C interface. This feature has been retained for backwards compatibility. The recommended method of generating a VCD file is to open a database with the `database -open -vcd` command and to probe objects to the database with the `probe -vcd` command. See the appendix called “Generating a VCD File Using CFC Routines” in the *NC VHDL Simulator Help* for more information. See “[call](#)” on page 504 for details on the `call` command.

See “[Generating an Extended Value Change Dump \(EVCD\) File](#)” on page 463 for more information on EVCD databases.

## Displaying Information About Databases

- If you are using the Tcl command-line interface, use the `database` command with the `-show` modifier to display information about databases.

Syntax:

```
database -show [{dbase_name | pattern} ...]
```

- If you are using the SimVision analysis environment, select *Show—Databases*.

## Disabling a Database

- If you are using the command-line interface, use the `database` command with the `-disable` modifier to temporarily disable a database.

Syntax:

```
database -disable {dbase_name | pattern} ...
```

- If you are using the SimVision analysis environment, enter the `database -disable` command at the prompt in the I/O region of the SimControl window.

## Enabling a Database

- If you are using the command-line interface, use the `database` command with the `-enable` modifier to enable a previously disabled database.

Syntax:

```
database -enable {dbase_name | pattern} ...
```

- If you are using the SimVision analysis environment, enter the `database -enable` command at the prompt in the I/O region of the SimControl window.

## Closing a Database

- If you are using the Tcl command-line interface, use the `database` command with the `-close` modifier to close a database.

Syntax:

```
database -close {dbase_name | pattern} ...
```

- If you are using the SimVision analysis environment, select *File–SHM Database–Close* to close a database. On the Close Database form, select the SHM database that you want to close and click the *OK* button.

To close a VCD or an EVCD database, enter the `database -close` command at the prompt in the I/O region of the SimControl window.

## Setting and Deleting Probes

You can save the values of objects to a database by probing them. The values contained in the database can be viewed using a waveform viewing tool.

### Setting a Probe

You can probe objects to the following kinds of databases:

- SHM (for Verilog, VHDL, or mixed-language)
- VCD (for Verilog, VHDL, or mixed-language)
- EVCD (for Verilog or VHDL)

If you are using the Tcl command-line interface, use the `probe` command with the optional `-create` modifier to create probes.

The basic syntax of the `probe` command is as follows:

```
probe [-create] [ {object | scope_name} ... ]  
      { -shm | -vcd | -evcd | -database dbase_name }
```

**Note:** For Verilog, you cannot use the `probe -create -evcd` command to probe Verilog signals to an EVCD database. Use the `$dumpports` system task in your Verilog code.

The `-create` modifier can be followed by an argument that specifies:

- The object(s) to be traced
- The scope(s) to be traced
- A combination of object(s) and scope(s) to be traced

If you do not specify an argument, the current debug scope is assumed, but you must include an option that specifies the objects you want to include in the trace (`-all`, `-inputs`, `-outputs`, or `-ports`).

You must include an option to specify the database into which values are dumped. Use one of the following options:

- `-database dbase_name`

Send the probe to the specified database. The database must already exist.

■ **-shm**

Send the probe to the default SHM database. If no default database is open, *ncsim* opens a default database called `ncsim.shm`.

■ **-vcd**

Send the probe to the default VCD database. If no default database is open, *ncsim* opens a default database called `ncsim.vcd`.

■ **-evcd**

Send the probe to the default EVCD database. If no default database is open, *ncsim* opens a default database called `ncsim.evcd`.

See “[probe](#)” on page 571 for details on the `probe` command and for information on probing Verilog and VHDL objects to different kinds of databases.

If you are using the SimVision analysis environment to probe objects to an SHM database, select *Set—Probe* to set a probe. If you are probing objects to a VCD or EVCD database, enter the `probe` command in the I/O region of the SimControl window.

See “[Setting Probes](#)” in the *SimVision Analysis Environment User Guide* for an example.

**Note:** Only objects that have read access are probed. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details on specifying access to simulation objects.

## Displaying Information About Probes

- If you are using the Tcl command-line interface, use the `probe` command with the `-show` modifier to display information about the probes that you have set.

Syntax:

```
probe -show [probe_name ...] [-database db_name]
```

- If you are using the SimVision analysis environment, select *Show—Probes* to display information about probes.

## Disabling a Probe

- If you are using the Tcl command-line interface, use the `probe` command with the `-disable` modifier to temporarily disable an SHM probe.

Syntax:

```
probe -disable probe_name [probe_name ...]
```

- If you are using the SimVision analysis environment, select the *Show—Probes* command to display the Debug Settings form and then click on the toggle button next to the SHM probe that you want to disable.

You can disable SHM probes individually at any time.

You cannot disable VCD and EVCD probes individually. Use `database -disable` to disable all VCD or EVCD probes. (See “[database](#)” on page 517.)

## Enabling a Probe

- If you are using the Tcl command-line interface, use the `probe` command with the `-enable` modifier to enable an SHM probe that was previously disabled.

Syntax:

```
probe -enable probe_name [probe_name ...]
```

- If you are using the SimVision analysis environment, use the *Show—Probes* command to display the Debug Settings form and then click on the toggle button next to the probe that you want to enable.

You cannot enable VCD and EVCD probes individually. Use `database -enable` to enable all VCD or EVCD probes. (See “[database](#)” on page 517.)

## Deleting a Probe

- If you are using the Tcl command-line interface, use the `probe` command with the `-delete` modifier to delete a probe.

Syntax:

```
probe -delete probe_name [probe_name ...]
```

- If you are using the SimVision analysis environment, use the *Show—Probes* command to display the Debug Settings form. Select the probe that you want to delete and then click the *Delete* button.

You can delete SHM probes at any time.

VCD and EVCD probes can only be deleted at the time the VCD or EVCD database is created. Once the simulation is advanced, `ncsim` writes the VCD or EVCD header to the file and no modifications to the probes are possible.

## Traversing the Model Hierarchy

The NC-Verilog simulator supports hierarchical designs by allowing models to be embedded within other models. Levels of hierarchy in a design are called *scopes*. To create a scope, you nest objects within design units by instantiating them. Instantiation allows one design unit to incorporate a copy of another into itself.

### Path Names

Each scope in a design hierarchy has a unique hierarchical path name. For Verilog, elements in the path name are separated by a period ( . ). Path names can be:

- Fully specified from the top level of the hierarchy. Full path names begin with the name of a Verilog top-level module. For example:

```
top.board.counter
top.vending.drinks.count_cans.in1
```
- Relative to the current debug scope. For example, if the current debug scope is `top.board`, the path name `counter` refers to a scope within the scope `board`, which is within the top-level module `top`.

See “[Path Names and Mixed-Language Designs](#)” on page 408 for information on how to specify path names for mixed Verilog/VHDL designs.

### Setting the Debug Scope

You traverse the model hierarchy by setting the scope to an instantiated object. If you are using the Tcl command-line interface, use the `scope -set` command. For example, if the current debug scope is the top level, and you want to scope down one level to a scope called `board`, use the following command:

```
% scope -set board
```

If you are at the top level and want to scope down to a scope within `board` called `counter`, use the following command:

```
% scope -set board.counter
```

You can specify a full path name from any debug scope. For example, if the current scope is `board:counter`, you can scope up to the top level (module `top`) with the following command:

```
% scope -set top
```

See “[scope](#)” on page 608 for details on using the `scope` command.

## **NC-Verilog Simulator Help**

### Debugging Your Design

---

If you are using the SimVision analysis environment, there are several ways to traverse the design hierarchy using the SimControl window or the Navigator. See “[Setting the Debug Scope](#)” in the *SimVision Analysis Environment User Guide* for examples of using SimControl. See “[The Navigator](#)” in the *SimVision Analysis Environment User Guide* for details on using the Navigator.

## Setting Breakpoints

You can interrupt the simulation by setting breakpoints.

For Verilog, you can set four kinds of breakpoints:

- Condition breakpoints. See “[Setting a Condition Breakpoint](#)” on page 427.
- Line breakpoints. See “[Setting a Source Code Line Breakpoint](#)” on page 428.
- Object breakpoints. See “[Setting an Object Breakpoint](#)” on page 429.
- Time breakpoints. See “[Setting a Time Breakpoint](#)” on page 430.

For VHDL, you can set the breakpoint types listed above plus:

- Delta breakpoints. See “[Setting a Delta Breakpoint](#)” on page 431.
- Process breakpoints. See “[Setting a Process Breakpoint](#)” on page 431.
- Subprogram breakpoints. See “[Setting a Subprogram Breakpoint](#)” on page 432.

## Setting a Condition Breakpoint

You can stop the simulation when a specified condition is true by setting a condition breakpoint. This type of breakpoint is particularly useful when you want to stop the simulation at the instant when a signal has been set to an incorrect value.

A condition breakpoint triggers when any object referenced in the conditional expression changes value (wires, signals, registers, and variables) or is written to (memories) and the expression evaluates to true (nonzero).

- If you are using the Tcl command-line interface, use the `stop` command with the `-condition` option to set a condition breakpoint.  
See “[stop](#)” on page 624 for details on using the `stop` command and for examples of setting breakpoints.
- If you are using the SimVision analysis environment, select *Set—Breakpoint—Condition* to set a condition breakpoint.  
See “[Setting a Condition Breakpoint](#)” in the *SimVision Analysis Environment User Guide* for an example of setting a condition breakpoint using the SimControl window.

A condition breakpoint takes a Tcl expression as an argument. See “[Tcl Expressions as Arguments](#)” on page 641 for details on the syntax of these expressions.

The simulator does not support breakpoints on individual bits of registers. If a bit-select of a register appears in the expression, the simulator stops and evaluates the expression when any bit of that register changes value. The same holds true for compressed wires.

Objects included in a conditional expression must have read access. An error is printed if the object does not have read access. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details on specifying access to simulation objects.

See “[Disabling, Enabling, Deleting, and Displaying Breakpoints](#)” on page 433 for more information on breakpoints.

## Setting a Source Code Line Breakpoint

You can stop the simulation at a specified line in the source code by setting a source code line breakpoint. This type of breakpoint is usually set when you want to simulate to a certain point and then single-step through lines of code.

You cannot set a line breakpoint unless you have compiled with the `-linedebug` option. (See [\\_linedebug](#) for details on using this option.)

To set a line breakpoint:

- If you are using the Tcl command-line interface, use the `stop` command with the `-line` option.  
See “[stop](#)” on page 624 for details on using the `stop` command and for examples of setting breakpoints.
- If you are using the SimVision analysis environment, select *Set—Breakpoint—Line* to set a line breakpoint.  
See “[Setting a Line Breakpoint](#)” in the *SimVision Analysis Environment User Guide* for an example of setting a line breakpoint using the SimControl window.

See “[Disabling, Enabling, Deleting, and Displaying Breakpoints](#)” on page 433 for more information on breakpoints.

## Setting an Object Breakpoint

You can stop the simulation when a specified object changes value (wires and signals) or when it is written to (registers, memories, variables) by setting an object breakpoint. This type of breakpoint is usually set when you want the simulation to stop every time the signal changes value or when you want to see the value of signals when some condition is true (for example, on every positive edge of the clock).

To set an object breakpoint:

- If you are using the Tcl command-line interface, use the `stop` command with the `-object` option.  
See “[stop](#)” on page 624 for details on using the `stop` command and for examples of setting breakpoints.
- If you are using the SimVision analysis environment, select *Set—Breakpoint—Object* to set an object breakpoint.  
See “[Setting an Object Breakpoint](#)” in the *SimVision Analysis Environment User Guide* for an example of setting an object breakpoint using the SimControl window.

The object specified as the argument must have read access for the breakpoint to be created. An error is printed if the object does not have read access. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details on specifying access to simulation objects.

By default, all vectors are compressed, and a `stop -object` command can only be used on the entire object. You must elaborate the design with the `-expand` option (`ncelab -expand`) in order to set a breakpoint on a subelement of a vector Verilog wire or VHDL signal.

You cannot set a breakpoint on an object in a VHDL subprogram.

## Setting a Time Breakpoint

You can stop the simulation at a specified time by setting a time breakpoint. The time can be absolute or relative (the default). Absolute time breakpoints are automatically deleted after they trigger. Relative time breakpoints are periodic, stopping, for example, every 10 ns.

This type of breakpoint is usually set when you want to advance the simulation to a certain time point before beginning to debug or when you want to stop the simulation at regular intervals to examine signal values.

To set a time breakpoint:

- If you are using the Tcl command-line interface, use the `stop` command with the `-time` option.  
See “[stop](#)” on page 624 for details on using the `stop` command and for examples of setting breakpoints.
- If you are using the SimVision analysis environment, select *Set—Breakpoint—Time* to set a time breakpoint.  
See “[Setting a Time Breakpoint](#)” in the *SimVision Analysis Environment User Guide* for an example of setting a time breakpoint using the SimControl window.

See “[Disabling, Enabling, Deleting, and Displaying Breakpoints](#)” on page 433 for more information on breakpoints.

## Setting a Delta Breakpoint

For VHDL, you can stop the simulation when the simulation delta cycle count reaches a specified delta cycle. To set a delta breakpoint:

- If you are using the Tcl command-line interface, use the `stop` command with the `-delta` option.  
See “[stop](#)” on page 624 for details on using the `stop` command and for examples of setting breakpoints.
- If you are using the SimVision analysis environment, select *Set—Breakpoint—Delta* to set a delta breakpoint.  
See “[Setting a Delta Breakpoint](#)” in the *SimVision Analysis Environment User Guide* for an example of setting a delta breakpoint using the SimControl window.

See “[Disabling, Enabling, Deleting, and Displaying Breakpoints](#)” on page 433 for more information on breakpoints.

## Setting a Process Breakpoint

For VHDL, you can stop the simulation when a named process starts executing or resumes executing after a wait statement.

**Note:** You must compile with the `-linedebug` option to enable the setting of source line and process breakpoints.

To set a process breakpoint:

- If you are using the Tcl command-line interface, use the `stop` command with the `-process` option.  
See “[stop](#)” on page 624 for details on using the `stop` command and for examples of setting breakpoints.
- If you are using the SimVision analysis environment, select *Set—Breakpoint—Process* to set a process breakpoint.  
See “[Setting a Process Breakpoint](#)” in the *SimVision Analysis Environment User Guide* for an example of setting a process breakpoint using the SimControl window.

See “[Disabling, Enabling, Deleting, and Displaying Breakpoints](#)” on page 433 for more information on breakpoints.

## Setting a Subprogram Breakpoint

For VHDL, you can stop the simulation when a named subprogram (procedure or function) starts executing.

**Note:** You must compile with the `-linedebug` option to enable the setting of source line, process, and subprogram breakpoints.

To set a subprogram breakpoint, use the `stop` command with the `-subprogram` option.

See “[stop](#)” on page 624 for details on using the `stop` command and for examples of setting breakpoints.

In the current version, you cannot set a subprogram breakpoint using the *Set* menu on the SimControl window. If you are using the SimVision analysis environment, enter the `stop` command in the simulator I/O region.

See “[Disabling, Enabling, Deleting, and Displaying Breakpoints](#)” on page 433 for more information on breakpoints.

## Disabling, Enabling, Deleting, and Displaying Breakpoints

After setting breakpoints, you can display information on breakpoints, disable breakpoints, enable previously disabled breakpoints, and delete breakpoints.

- If you are using the command-line interface, use the `stop` command with the `-show`, `-disable`, `-enable`, or `-delete` modifier. The argument to these modifiers can be:
  - a break name or a list of break names
  - a pattern
  - The asterisk ( `*` ) matches any number of characters
  - The question mark ( `?` ) matches any one character
  - `[characters]` matches any one of the characters
  - Any combination of literal break names and patterns
- See “[stop](#)” on page 624 for details on using the `stop` command.
- If you are using the SimVision analysis environment, select *Show—Breakpoints* to open the Debug Settings form. Click on the button next to a breakpoint to disable or enable a breakpoint. To delete a breakpoint, select the breakpoint and then click the *Delete* button.

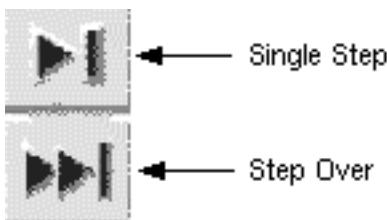
See “[Disabling, Enabling, Deleting, and Displaying Breakpoints](#)” in the *SimVision Analysis Environment User Guide* for an example.

## Stepping Through Lines of Code

You can examine the order in which the simulator executes the statements in your model by stepping through the simulation line by line.

**Note:** You cannot single step one line at a time or set line breakpoints in a particular design unit unless you have compiled that unit with the `-linedebug` option. If you have compiled the unit without this option, the `run -step` or `run -next` commands will run the simulation until the next point where it can stop. If execution is passed to a unit that was compiled with `-linedebug`, full single stepping is resumed.

- If you are using the Tcl command-line interface:
  - Use `run -step` to simulate to the next executable line of code in any scope. This command runs one statement, stepping into subprogram calls.
  - Use `run -next` to run one statement, stepping over any subprogram calls.
- See “[run](#)” on page 598 for details on using the `run` command.
- If you are using the SimVision analysis environment, select *Run—Step* or *Run—Next*. You can also click on the *Single Step* or *Step Over* buttons on the Tool Bar.



See “[Stepping Through the Simulation](#)” in the *SimVision Analysis Environment User Guide* for an example.

## Forcing and Releasing Signal Values

You can ask “What if” questions about your model by interactively forcing objects to desired values and seeing if the patch fixes the problem. If it does, you can then edit your source file to incorporate the change.

The object that is being forced must have write access. An error is printed if it does not. To specify write access, use the `-access` or `-afile` option when you elaborate the design with `ncelab`. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details on specifying access to simulation objects.

- If you are using the Tcl command-line interface, use the `force` command to set a specified object to a given value and force it to retain that value until it is released with a `release` command or until another force is placed on it.

See “[force](#)” on page 551 and “[release](#)” on page 592 for details on using these commands.

- If you are using the SimVision analysis environment, select *Set—Force* to force a signal to a given value. To release a signal, position the pointer over the signal, click the right mouse button, and select *Release Force* to release the signal.

See “[Forcing and Releasing Signal Values](#)” in the *SimVision Analysis Environment User Guide* for an example.

## Depositing Values to Signals

Besides forcing an object to a desired value using the `force` command (*Set—Force* in SimVision), another way to ask “What if” questions about your model as you debug is to interactively deposit a value to a specified object.

When you deposit a value to an object, behaviors that are sensitive to value changes on the object run when the simulation resumes, just as if the value change was caused by the Verilog or VHDL code.

You can deposit a value to an object immediately, at a specified time in the future, or after a specified delay. You can also specify that you want to deposit the value after an inertial delay or after a transport delay. A deposit without a delay is similar to a force in that the specified value takes effect and propagates immediately. However, it differs from a force in that future transactions on the signal are not blocked.

For VHDL, you can deposit to ports, signals, and variables if no delay is specified. If a delay is specified, you cannot deposit to variables or to signals with multiple sources.

For Verilog, you can deposit to ports, signals (wires and registers), and variables.

The object that you want to deposit a value to must have write access. An error is printed if it does not. To specify write access, use the `-access` or `-afile` option when you elaborate the design with `ncelab`. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details on specifying access to simulation objects.

To deposit a value to an object:

- If you are using the Tcl command-line interface, use the `deposit` command to set a specified object to a given value.  
See “[deposit](#)” on page 528 for details on using the `deposit` command.
- If you are using the SimVision analysis environment, select *Set—Deposit* to deposit a value to a signal.  
See “[Depositing Values to Signals](#)” in the *SimVision Analysis Environment User Guide* for an example.

## Displaying Information About Simulation Objects

You can display information about a simulation object, including its declaration.

- If you are using the Tcl command-line interface, use the `describe` command.  
See “[describe](#)” on page 532 for details on using this command.
- If you are using the SimVision analysis environment, select *Show—Description* and fill in the Show Description form with the name of the object.  
See “[Displaying Information About Simulation Objects](#)” in the *SimVision Analysis Environment User Guide* for an example using SimControl.

## Displaying the Drivers of Signals

You can display a list of all of the contributors to the value of a specified signal(s).

- If you are using the Tcl command-line interface, use the `drivers` command. For example:

```
ncsim> drivers board.count
```

See “[drivers](#)” on page 537 for details on using this command and for examples of the report format for Verilog signals and for VHDL signals.

You can use the `scope -drivers [scope_name]` command to display the drivers of each object that is declared within a specified scope. See “[scope](#)” on page 608 for details on the `scope` command.

- If you are using the SimVision analysis environment, select *Show—Drivers* and fill in the Show Drivers form with the name of the signal(s) for which you want to display driver information.

See “[Displaying Signal Drivers](#)” in the *SimVision Analysis Environment User Guide* for an example.

The `drivers` command cannot find the drivers of a wire or register unless the object has connectivity access. However, even if you have specified access to the object, its drivers may have been collapsed, combined, or optimized away. In this case, the output of the `drivers` (or *Show—Drivers*) command might indicate that the object has no drivers. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details on specifying access to simulation objects.

## Checking for Bus Contention and Bus Float Conditions

The NC-VHDL simulator lets you check for bus contention and bus float conditions in your design. You can perform these two design checks only on VHDL std\_logic bus signals (a signal that has multiple drivers). Checks cannot be applied on signals that are declared in a VITAL Level0 scope.

### ■ Bus contention detection

Bus contention checking detects bus fights on nodes that have multiple drivers. If more than one driver of a std\_logic bus signal drives a non-Z value for more than a specified time limit, the condition is reported as a bus contention.

If the simulator detects a bus contention, it issues an error message. The message includes the name of the bus signal on which the contention was detected, the names of the drivers and their current values, the time at which the contention started, and the time at which the time window was exceeded.

### ■ Bus float detection

Bus float checking detects nodes that are in the high impedance state for a time that exceeds a specified time limit. If no driver of a std\_logic bus signal is driving a non-Z value for more than a specified time limit, the condition is reported as a bus float.

If the simulator detects a bus float, it issues an error message. The message includes the bus signal name, the time at which the float started, and the time at which the time window was exceeded.

Use the `Tcl check` command to check for bus contention/bus float conditions on a specified VHDL bus signal(s). Use the `-contention` option to create a bus contention check, or use the `-float` option to create a bus float check.

The following example shows how to use the `check` command to create a bus contention check. If more than one driver of bus signal `sig1` is driving a non-Z value and if `sig1` is active for more than 10 ns, the simulator reports a bus contention message. If the duration of the bus contention is 10 ns or less, the condition is not reported.

```
ncsim> check -contention sig1 -delay 10 ns
```

The following example shows how to use the `check` command to create a bus float check. If no driver is driving bus signal `sig1` for more than 10 ns, the simulator reports a bus float message. If the duration of the bus float is 10 ns or less, the condition is not reported.

```
ncsim> check -float sig1 -delay 10 ns
```

You can set the time limit for bus contention or bus float checks in two ways:

- By including the `-delay` option on a `check -contention` or a `check -float` command, as shown in the previous examples.
- By issuing a `check -delay` command. This sets a default time limit for subsequent checks. In the following example, the `check -delay` command sets a default time limit of 10 ns for the subsequent bus contention check.

```
ncsim> check -delay 10 ns  
ncsim> check -contention sig1
```

If you do not specify a time limit using a `check -delay` command or in the `check -condition` or `check -float` command, the default time limit value is 1 fs.

Changing the default time limit using a `check -delay` command does not affect existing checks.

The bus contention or bus float time limit cannot be smaller than, or more precise than, the unit of simulation.

The `check` command checks for bus contention and bus float conditions at the following times:

- If any of the drivers of the bus signal change.
- At the end of the simulation. This is to take care of a situation when a contention or float occurs on a bus signal sometime during the simulation, and after that, none of the drivers of this bus signal change. If the check was done only on a driver change, a bus contention or bus float on such a bus signal would go undetected.

See “[check](#)” on page 508 for details on the `check` command.

## Displaying Waveforms with SimVision Waveform Viewer

Use SimVision Waveform Viewer to display and analyze waveforms that are stored in an SHM (SST2) or a VCD database.

Only SHM (SST2) databases are supported for interactive waveform display.

This section tells you how to open a database, how to probe signals, and how to invoke SimVision Waveform Viewer. See the *SimVision Waveform Viewer User Guide* for details on using SimVision Waveform Viewer.

**Note:** The objects that you want to probe to an SHM or VCD database must have read access. By default, objects in the design are not given read access. Use the `_access +r` option or the `_afile access_file` option when you elaborate the design (`+ncaccess+r` or `+ncafile+access_file` if you are running `ncverilog`) to provide read access.

### Creating an SHM Database and Probing Signals

You can open a database, probe signals, and save the results in the database by entering Tcl commands at the prompt or by using the graphical user interface.

See “[Managing Databases](#)” on page 418 for details on opening a database. That section also contains information on displaying information about databases and on disabling, enabling, and closing a database.

See “[Setting and Deleting Probes](#)” on page 421 for details on probing signals.

You also can use a set of system tasks to open an SHM database, probe signals, and save the results in the waveform database. You must enter the system tasks into the Verilog description prior to simulation. The system tasks are:

---

System task	Description
<code>\$shm_open( );</code>	Opens a simulation database.
<code>\$shm_probe( );</code>	Specifies the signals whose simulation value changes are entered into the simulation database.
<code>\$shm_close;</code>	Closes a simulation database.

---

Example:

```
initial
begin
    $shm_open( "waves.shm" );
    $shm_probe();
    #1 $stop; // stop simulation at time 1
end
```

**Note:** For backward compatibility with Verilog code that contains calls to the Signalscan tasks for recording data during a Verilog simulation (\$recordvars, \$recordfile, \$recordsetup, and so on), Cadence has implemented these tasks as system tasks native to the simulator. You can keep these calls in your Verilog code, and they will function as intended. See [“Using \\$recordvars and Related Tasks”](#) on page 446 for details.

## Opening a Database with `$shm_open`

Use the `$shm_open` system task to open an SHM database.

The syntax of the `$shm_open` system task is as follows:

```
$shm_open ( "db_name" , is_sequence_time , db_size , is_compression );
```

All four arguments are optional. The arguments are:

### **"*db\_name*"**

Specifies the filename of the simulation database. If you do not specify the database name, a database called `waves.shm` is opened in the current directory.

### ***is\_sequence\_time***

Dumps all value changes to the database.

By default, when probing to an SHM database, the simulator discards multiple value changes for an object during one simulation time and dumps only the final value at the end of that simulation time. Specify 1 for the `is_sequence_time` argument if you want to dump all value changes to the SHM database. You can then use SimVision Waveform Viewer to expand a single moment of simulation time to show the sequence of value changes that occurred at that time. For example:

```
$shm_open( "mywaves.shm" , 1 , , );
```

***db\_size***

Specifies the maximum size (in bytes) of the transition file (.trn file).

The simulator maintains the size limit of the transition file by discarding the earliest recorded values as new values are dumped, such that the database always contains the most recent values for each probed object.

When the size limit is exceeded, the waveform window displays an "unknown" value for each object from the beginning of the simulation to the time of the first non-discarded value.

The SHM database uses approximately 2.5Mb of disk space, even if you specify a lower limit. However, the database size will not exceed the limit if the limit is greater than 2.5Mb. For example:

```
$shm_open("mywaves.shm", 1, 250000, );
```

***is\_compression***

Compresses the SHM database to reduce its size. The default setting is 0. Specify 1 to compress the database file. For example:

```
$shm_open("mywaves.shm", 1, , 1);
```

If you open an SHM database with the \$shm\_open system task in your Verilog code, the name of the database that is created is preceded by an underscore character. For example, the following system task opens a database called \_waves.shm.

```
$shm_open("waves.shm");
```

## Probing Signals with \$shm\_probe

The \$shm\_probe system task lets you specify the signals whose value changes you want to record in your SHM database and lets you specify the nodes at which value changes are recorded.

The syntax for the \$shm\_probe system task is as follows:

```
$shm_probe( [scope1, "nodeSpecifier1", scope2, "nodeSpecifier2", ... ] )
```

The arguments to \$shm\_probe are optional, but the parentheses are not.

If you do not specify any arguments, \$shm\_probe writes the value changes that occur at all inputs, outputs, and inouts in the current scope to your SHM database.

The arguments to this task are as follows:

- *scope<sub>1</sub>, scope<sub>2</sub>, ...*

Specifies the scope or scopes whose signals you want to probe. If you do not specify a scope, \$shm\_probe records the signal changes that occur in the current scope.

- *"node\_specifier<sub>1</sub>", "node\_specifier<sub>2</sub>", ...*

Specifies one of five codes, called node specifiers, to indicate the nodes at which you want to record value changes for the specified signals.

Node specifiers apply to the specified scopes in order of appearance. That is, *node\_specifier<sub>1</sub>* applies to *scope<sub>1</sub>*, *node\_specifier<sub>2</sub>* applies to *scope<sub>2</sub>*, and so on. If a node specifier does not have a corresponding scope, it applies to the current scope. If a scope does not have a corresponding node specifier, \$shm\_probe records value changes at all inputs, outputs, and inouts in that scope.

The five node specifiers are:

---

<b>Node Specifier</b>	<b>Signals That Enter the Database</b>
"A"	All nodes (including inputs, outputs, and inouts) in the specified scope.
"S"	Inputs, outputs, and inouts of the specified scope, and in all instantiations below the specified scope, except nodes inside library cells.
"C"	Inputs, outputs, and inouts of the specified scope, and in all instantiations below the specified scope, including nodes inside library cells.
"AS"	All nodes (including inputs, outputs, and inouts) in the specified scope, and in all instantiations below it, except nodes inside library cells.
"AC"	All nodes (including inputs, outputs, and inouts) in the specified scope and in all instantiations below it, including nodes inside library cells.

---

The following examples show you how to use \$shm\_probe to choose the signals and nodes whose value changes you want to record in your SHM database:

- `$shm_probe( );`  
Record value changes at all inputs, outputs, and inouts in the current scope.
- `$shm_probe( "A" );`  
Record value changes at all nodes in the current scope.
- `$shm_probe(alu, adder);`  
Record value changes at all inputs, outputs, and inouts in the scopes alu and adder.
- `$shm_probe( "S" , top.alu, "AC" );`  
Record value changes at all inputs, outputs, and inouts in the current scope and below, excluding nodes in library cells. Also, record value changes at all nodes in the scope top.alu and in all scopes below top.alu, including nodes in library cells.

## Invoking SimVision Waveform Viewer

When working with waveforms, there are two basic use models:

- Simulate and generate a database. Then invoke the waveform tool to view the waveforms. You can also use the SimVision analysis environment in post-processing mode (PPE).
- Invoke the simulator with the SimVision analysis environment so that you can simulate and view the waveforms as the simulation progresses.

### Invoking SimVision Waveform Viewer to View a Post-Simulation Database

After you have created a database, you can invoke SimVision Waveform Viewer to view the waveforms.

- On UNIX, invoke SimVision Waveform Viewer with the following command:

```
% simvision [database_name]
```

For example:

```
% simvision  
% simvision waves.shm
```

If you do not specify a database:

1. Select *File—Open Database* in the Design Browser window to open the Open Database form.

2. Double-click on the directory that you want to browse.
3. Select the *.trn* file in the *File* list box.
4. Click the *OK* button.
5. Select the signals that you want to display in the Waveform window.
6. Click the *Waveform* button to add the signals to the Waveform window.

If you specify a database on the command line, select the signals that you want to display in the Waveform window and click the *Waveform* button.

- On Windows NT, invoke SimVision Waveform Viewer from the Start menu or by double-clicking on the transition (*.trn*) file, and then use the Design Browser to select the signals that you want to view.

### **Invoking SimVision Waveform Viewer for Interactive Waveform Display**

If you want to interactively debug your design using the waveform tool:

1. Open a database with the *File—Waveform Database—Open* command.

On the Open Database form, select the *open as default database* option and set other options.

2. Probe signals or scopes and invoke SimVision Waveform Viewer.

Select the signals or scopes that you want to probe and then select *Windows—Waveform*. You can also click the *Waveform View* button or select *Wave Trace* from the pop-up menu.

This displays the selected signals or scopes in the Waveform window with no values.

3. Simulate.

If you do not preselect objects, selecting *Windows—Waveform* or clicking the *Waveform View* button invokes SimVision Waveform Viewer. You can then probe signals and add them to the waveform window using the Design Browser.

A useful shortcut is to preselect the signals or scopes that you want to view in the waveform tool and then select *Windows—Waveform* (or click the *Waveform View* button or select *Wave Trace* from the pop-up menu). This opens a default database called *waves.shm*, probes the selected signals, and invokes SimVision Waveform Viewer with the selected signals displayed in the waveform window. If you use this shortcut, however, the Open Database form does not appear, and you cannot select any options that appear on that form, such as the option to record all events or the option to specify a maximum database size.

## Using \$recordvars and Related Tasks

If you have Verilog code that contains calls to the Signalscan \$recordvars and related tasks (\$recordfile, \$recordsetup, and so on), you can use these calls to record data in an SHM (SST2) database. The PLI tasks that were used for recording data during a Verilog simulation have been implemented as system tasks native to the simulator. That is, it is not necessary to write PLI code and to link this into the simulator.

The following Signalscan PLI tasks have been implemented as system tasks native to the simulator:

- \$recordvars
- \$recordfile
- \$recordsetup
- \$recordon/\$recordoff
- \$recordclose
- \$recordabort
- \$signalscan

Only the tasks that are used for recording data have been implemented as system tasks. The following Signalscan tasks for interfacing with Signalscan from your Verilog code are not supported, and using them will generate warning messages: \$signalscanconnect, \$signalscancommand, \$signalscankill, \$signalscanabort.

In addition, some options to the \$record\* tasks have not been implemented. These are the options that have to do with writing incremental files (incsize, incetime, inccpu, incfiles, summary, and nosummary). Using these options will generate warning messages.

If you must use the \$signalscan\* tasks listed above or the incremental file options, you can revert to using the PLI interface support. There are two easy ways to do this:

- Add the following path to your definition of the LD\_LIBRARY\_PATH (Solaris) or SHLIB\_PATH (HP) variable:  
*your\_install\_dir/tools/simvisdai/lib*
- Create a link to the PLI interface library with the following command:  

```
ln -s your_install_dir/tools/simvisdai/lib/record-scb.so
      your_install_dir/tools/lib/.
```

There are a few differences between the database that is dumped using the new built-in system tasks and the database that is dumped using the PLI interface support. These differences include the following:

- The database dumped using the PLI interface support includes the highconns for ports. These are not always included in the database that is dumped using the new built-in system tasks.
- The database dumped using the PLI interface support includes continuous assignments that are not always included in the database that is dumped using the new built-in system tasks.
- For primitive terminals, the database that is dumped using the new built-in system tasks always dumps the signal to which the terminal is connected.

## \$recordvars

The only system task required to record data to an SHM (SST2) database is \$recordvars.

Syntax:

```
$recordvars[ ("options")];
```

If you do not specify any options, value changes on all signals in the design hierarchy (with no driver or primitive information) are recorded.

Only one database may be written at a time, but you can add variables to be recorded to the database at any time by using another \$recordvars task.

The following table lists the options that you can use with \$recordvars. Options apply to all following variables and scopes in the call, or to the default scopes if none are specified.

---

Option	Effect	Default
"depth= <i>n</i> "	Limit the depth if a scope is specified. If "depth=1", no child scopes are included.	"depth=0"
"drivers"	Record drivers (an output terminal of a primitive). Drivers are recorded for all recorded variables that have more than one driver.	"nodrivers"
"primitives"	Record primitives. For all scopes that are recorded, record their primitives in addition to their variables.	"noprimitives"

## NC-Verilog Simulator Help

### Debugging Your Design

---

<b>Option</b>	<b>Effect</b>	<b>Default</b>
"nocells"	Do not record variables within a cell, or within any scopes below the cell.  By default, modules defined in a library are cells and other modules are not cells. A non-library module can be defined as a cell using the Verilog 'celldefine directive.	"cells"
"noports"	Do not record port connectivity information.  This option is used primarily to work around a simulator defect that affects some designs. If this option is used, Signalscan does not display ports in different colors, the Schematic Tracer does not display port connectivity, and the Add Trace and Add Module Inputs commands do not display ports.	"ports"
"trace"	Record statement trace information.  If you use this option, you must also use the "sequence" option on either the \$recordfile or \$recordsetup task.  Recording statement trace information is independent of what variables you are recording. You must record variables in separate \$recordvars task statements. Do not specify other options in the same \$recordvars statement where you specify the "trace" option.	
Any variable	Record a variable. A variable can be a net, register, integer, time, real, or named event. For example, top.u1.u32.a.	
Any scope	Record a scope. By default, all variables within the scope and all variables in all child scopes are recorded in the database. Use "depth=n" to limit the depth.  A scope can be a module, task, function, or named block. For example, top.control.	All top-level modules and all subscopes.

---

If you open an SHM database with the `$record*` system task in your Verilog code, the name of the database that is created is preceded by an underscore character. For example, assuming that the top-level module is called `top`, the following system task opens a database called `_top`.

```
$recordvars;
```

The following system tasks open a database called `_results`.

```
$recordfile("results");
$recordvars;
```

This lets you interact with databases opened with `$record*` in the same way that you interact with databases that you open with the `database` command or with `$shm_open`. That is, you can use the `database` command to disable, enable, or display information about the databases.

The `$recordvars` task generates two output files:

- A design file (`.dsn`), which contains information about the design.

By default, the name of this file is `design_name-version_name.dsn`. For example, `top-1.dsn`.

- A transition file (`.trn`), which contains the transition information.

By default, the name of this file is `design_name-version_name-run_name.trn`. For example, `top-1-1.trn`.

Use the `$recordsetup` task to override the default `design_name`, `version_name`, and `run_name`.

**Note:** If you revert to using the PLI interface support, the file naming convention is the same as that described above if you do not include a `$recordfile` task to specify the name of the database. If you use `$recordfile` to specify a database name, the files are called `database_name.dsn` and `database_name.trn`. These files are overwritten every time the simulator is run. For example the following code generates `results.dsn` and `results.trn`:

```
$recordfile("results");
$recordvars;
```

### **Example 1:**

In the following example, no options, variables, or scopes are specified in the `$recordvars` call. All top-level modules are used by default, and all variables in the design are recorded.

```
module record;
    initial $recordvars;
```

```
endmodule
```

### **Example 2:**

In the following example:

- The first \$recordvars records all variables within scope `top.mod1` and all of its descendants, but records only the variables three levels deep for scope `top.mod2`.
- The second \$recordvars illustrates how options apply to all variables specified later in that \$recordvars task unless overridden. This task records driver information for variables in `mod1` and driver and primitive information for variables in `mod2`.
- The third \$recordvars records two explicitly named variables.

```
module record;  
initial  
begin  
    $recordvars(top.mod1, "depth = 3", top.mod2);  
    $recordvars("drivers", mod1, "primitives", mod2);  
    $recordvars(top.io.mux1.q0, top.io.mux2.q0);  
end  
endmodule
```

### **Example 3:**

In the following example:

- The first \$recordvars records three levels of variables within scope `top.mod1`. Drivers are also recorded if the variables have more than one driver. Scope `top.mod2` is not depth restricted and no driver information is recorded for variables in this scope.
- The second \$recordvars records driver information for variables in `mod1` and driver and primitive information for variables in `mod2`.
- The third \$recordvars records `top.middle.clock` and all variables in `module2` and its subscopes.
- The fourth \$recordvars records statement trace information. Recording statement trace information is independent of what variables you are recording. You must record variables in \$recordvars task statements, and specify the trace option in a separate \$recordvars statement.
- The \$recordsetup task in this example specifies the recording of sequence information. Sequence information is needed to correlate statements and transitions. If you collect trace information but do not collect sequence information, you will receive a warning message during simulation.

The recording of statement trace information is either on or off for the entire simulation. The \$recordon and \$recordoff statements have no effect on recording statement trace information. Other \$recordvars options, such as specifying depth or scopes, have no effect on how much statement trace information is recorded.

```
module record;
    initial
        begin
            $recordsetup("design = mydesign", "sequence");
            $recordvars("depth = 3", "drivers", top.mod1,
                       "depth = 0", "nodrivers", top.mod2);
            $recordvars("drivers", mod1, "primitives", mod2);
            $recordvars(top.middle.clock, module2);
            $recordvars("trace"); // Must be alone
        end
    endmodule
```

## \$recordfile

The \$recordfile task records basic design information and sets up the recording options for variables recorded with \$recordvars. This task is optional. If you use it, the task should be placed before the first \$recordvars task.

Syntax:

```
$recordfile( filename [, "options"] );
```

### *filename*

The name of the database. This can be a string enclosed in double quotes, or the name of a variable that contains the file name. Although not required, the extension .trn is recommended to identify the transition database. A .dsn file is also created with the same base name as the .trn file.

The following table lists the options that you can use with the \$recordfile task.

**Note:** The following \$recordfile options, all of which have to do with writing incremental files, are not supported. Using them will generate a warning message.

- “incsize = *size*”
- “inctime = *simtime*”
- “inccpu = *cputime*”
- “incfiles = *count*”

## NC-Verilog Simulator Help

### Debugging Your Design

---

- “summary[=*file*]” and “nosummary”

<b>Option</b>	<b>Effect</b>	<b>Default</b>
“wrapsize= <i>size</i> ”	<p>Limit the size of the .trn file before data is wrapped into another file.</p> <p>The size argument is a number followed by B (bytes), K (kilobytes), M (megabytes), or G (gigabytes). The default is M.</p> <p>When the transition data exceeds the specified size, the oldest transitions are overwritten by newer transitions. However, transitions are written to the file, and discarded from the file, in blocks of about 4-5 Mb. This means that the actual size of the database can be considerably larger than, or smaller than, the specified size.</p> <p>It is recommended that the maximum size be at least 10 Mb, if specified.</p>	
“sequence”	Save sequence information (the sequence in which events occurred). This is necessary for tracing.	“nosequence”
“compress”	Compress the database. Sequence information is not compressed.	“nocompress”

Example:

In the following example, a design file named adder-1.dsn and a transition file named adder-1-1.trn is created. The transition file is compressed. Sequence time is recorded in the database. You can record sequence information with either the \$recordfile or the \$recordsetup task.

You can use the "compress" and "sequence" options together, but only transition information is compressed; sequence information is not compressed.

```
$recordfile("adder", "compress", "sequence");  
$recordvars;
```

### **\$recordsetup**

The \$recordsetup task records basic design and hierarchy information and sets up the recording options for variables recorded with \$recordvars. This task is optional. If you use it, the task should be placed before the first \$recordvars task.

When \$recordsetup is called, the scope hierarchy is recorded in the design file immediately. However, primitives and variables are not recorded until \$recordvars is called.

Syntax:

```
$recordsetup( [ "options" ] );
```

The following table lists the options that you can use with the \$recordsetup task.

**Note:** The following \$recordsetup options, all of which have to do with writing incremental files, are not supported. Using them will generate a warning message.

- "incsize = *size*"
- "inctime = *simtime*"
- "inccpu = *cputime*"
- "incfiles = *count*"
- "summary[=*file*]" and "nosummary"

---

Option	Effect	Default
"design= <i>name</i> "	Create a name for the design.	Name of the first top scope found.

## NC-Verilog Simulator Help

### Debugging Your Design

---

<b>Option</b>	<b>Effect</b>	<b>Default</b>
"version= <i>name</i> "	Name this version of the design.	Next number (based on the files in the current directory or the directory specified with the "directory" option).
"run= <i>name</i> "	Name this particular simulation run.	Next number (based on the files in the current directory or the directory specified with the "directory" option).
"directory= <i>path</i> "	Specify the directory where the files will be saved. If the specified directory does not exist, it is created for you.	Current working directory.
"wrapsize= <i>size</i> "	Limit the size of the .trn file before data is wrapped into another file.  The size argument is a number followed by B (bytes), K (kilobytes), M (megabytes), or G (gigabytes). The default is M.  When the transition data exceeds the specified size, the oldest transitions are overwritten by newer transitions. However, transitions are written to the file, and discarded from the file, in blocks of about 4-5 Mb. This means that the actual size of the database can be considerably larger than, or smaller than, the specified size.  It is recommended that the maximum size be at least 10 Mb, if specified.	
"sequence"	Save sequence information (the sequence in which events occurred). This is necessary for tracing.	"nosequence"

Option	Effect	Default
"compress"	Compress the database. Sequence information is not compressed.	"nocompress"

### **Example 1:**

In the following example, a design file named `data/adder-1.dsn` is created. If `adder-1.dsn` already exists in the `data` directory, `adder-2.dsn` is created. A transition file named `data/adder-1-1.trn` is created. If this file already exists, a file called `adder-2-1.trn` is created.

```
module record;
initial
begin
$recordsetup("directory = data", "design = adder");
$recordvars;
end
endmodule
```

### **Example 2:**

In the following example, a design file named `data/adder-algo1.dsn` is created, or replaced if it exists. The database is compressed.

```
module record;
initial
begin
$recordsetup("directory = data", "design = adder", "version = algo1",
"compress");
$recordvars;
end
endmodule
```

### **\$recordon/\$recordoff**

Use the `$recordon` and `$recordoff` tasks to turn recording on or off, respectively. Recording can be turned on or off at selected times or based on conditions in Verilog.

The `$recordoff` task does not close the database file. Variable transitions are not recorded during the period where recording is off. All recorded variables are updated to their current values when recording is turned back on.

**Example:**

In the following example, the \$recordon and \$recordoff tasks are used to record variables for a portion of the total simulation time.

```
module record;
initial
begin
$recordvars;
$recordoff;
end
endmodule

module top;
reg clock;
initial
begin
#0 clock=0;
#100 clock=1;
#100 clock=1;
#100 clock=0; $recordon;
#100 clock=1;
#100 clock=0;
#100 clock=1; $recordoff;
#100 clock=0;
#100 clock=1;
end
endmodule
```

**\$recordclose**

Use the \$recordclose task to close an open database. This task stops the recording of data, flushes buffered data to the database, and closes the database.

**Example:**

```
$recordclose;
```

**\$recordabort**

Use the \$recordabort task to abort recording to a database that is no longer wanted. Any buffered information not yet written to the database is discarded, and the database is deleted. Any current interactive connection to Signalscan is also aborted.

Example:

```
$recordabort;
```

### **\$signalscan**

You can launch Signalscan from your Verilog code with the \$signalscan task. This task is used to interactively view simulation results in Signalscan while the simulation is running. All variables being recorded to the database are available for viewing. Conversely, a variable must be recorded in the database in order to view it with Signalscan. This means that there must be at least one call to \$recordvars in order for \$signalscan to be useful.

Syntax:

```
$signalscan( [ "path = path_to_signalscan_executable" ] [ , "arguments" ] );
```

The path to the Signalscan executable is optional. If you do not specify a path, the PATH environment variable is used to find the executable, which must be named signalscan.

You can also pass Signalscan arguments as parameters to the \$signalscan task. The parameter is a string enclosed in double quotes.

There can be only one interactive Signalscan connection at a time for each simulation. If you exit Signalscan or the simulator, or use the \$recordclose or \$recordabort tasks, the interactive connection is closed. You can then call the \$signalscan task again to start a new interactive connection.

Example:

In the following example, the \$signalscan task specifies an absolute path to the Signalscan executable. It also includes an argument to load a Do-File.

```
module record;
  initial
    begin
      $recordvars;
      $signalscan("path=/usr/tools/signalscan-6.2/signalscan", "-do my.do");
    end
  endmodule
```

## Generating a Value Change Dump (VCD) File

A value change dump (VCD) file is an ASCII file that contains information about value changes on selected variables in the design. The file contains header information, variable definitions, and the value changes for all specified variables.

The Verilog LRM describes two types of VCD files:

- Four-state—Represents variable changes in 0, 1, x, and z with no strength information.
- Extended—Represents variable changes in all states and with strength information.

This section tells you how to generate a four-state VCD file. See “[Generating an Extended Value Change Dump \(EVCD\) File](#)” on page 463 for information on generating an extended VCD file.

In NC-Verilog, there are two ways to generate a four-state VCD file:

- Open a VCD database with the `Tcl` database `-open -vcd` command, and then probe signals to the database with the `probe -create -vcd` command.
- Use the VCD system tasks in your Verilog code.

**Note:** You can only dump objects that have read access. If you specify a scope as an argument to the `probe` command, or as an argument to the `$dumpvars` system task, objects within that scope that do not have read access are excluded from the dump, and the simulator prints a warning message. If you specify an individual variable and that object does not have read access, the simulator prints an error message. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details on specifying access to simulation objects.

### Generating a VCD File with `Tcl` Commands

To generate a VCD file by using `Tcl` commands:

1. Open a VCD database with the `database` command. The syntax is as follows:

```
database [-open] dbase_name -vcd  
[-default]  
[-into filename]  
[-maxsize max_byte_size]  
[-timescale timescale_value]
```

The following command opens a VCD database named vcddb. The filename is verilog.dump. The -timescale option sets the \$timescale value in the VCD file to 1 ns. Value changes in the VCD file are scaled to 1 ns.

```
ncsim> database -open -vcd vcddb -into verilog.dump -timescale ns  
Created VCD database vcddb
```

See “[database](#)” on page 517 for details on the `database` command.

**2. Probe objects to the database with the `probe` command.** The syntax is as follows:

```
probe [-create] [{object | scope_name}... ] {-vcd | -database dbase_name}  
[-all]  
[-depth {n | all | to_cells}]  
[-inputs]  
[-name probe_name]  
[-outputs]  
[-ports]  
[-screen [-format format_string] [-redirect filename] objects]
```

The optional `-create` modifier can be followed by an argument that specifies:

- The object(s) to be traced
- The scope(s) to be traced
- A combination of object(s) and scope(s) to be traced

If you do not specify an argument, the current debug scope is assumed, but you must include an option that specifies which objects to include in the trace (`-all`, `-inputs`, `-outputs`, or `-ports`).

If more than one database is open, you must include an option to specify the database into which you want to dump values. You can do this either by specifying a database name with the `-database` option or by using the `-vcd` option to send the probe to the default VCD database. If no default database is open, the simulator opens a default database called `ncsim.vcd`.

The following `probe` command creates a probe on all ports in the scope `top.counter`. Data is sent to the default VCD database.

```
ncsim> probe -create -vcd top.counter -ports  
Created probe 1
```

See “[probe](#)” on page 571 for details on the `probe` command.

You can use the Tcl `database` and `probe` commands to generate a VCD file for a mixed-language design. See “[Generating a Value Change Dump \(VCD\) File for a Mixed-Language Design](#)” on page 410 for an example.

## Generating a VCD File with VCD System Tasks

Several system tasks can be inserted in the source description to create a four-state VCD file. NC-Verilog supports all of the value change dump system tasks specified in the IEEE Verilog Language Reference Manual. This section summarizes the system tasks.

See the *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language* (IEEE Std 1364-1995 or 1364-2001) for details on these VCD system tasks, and for information on the syntax and format of the VCD file.

### VCD System Tasks

NC-Verilog supports all of the value change dump system tasks specified in the IEEE Verilog Language Reference Manual. These system tasks are:

- `$dumpfile ("filename");`

Specifies the name of the VCD file. If you do not use this system task to specify a file name, the simulator creates a file called `verilog.dump` by default.

- `$dumpvars;`

Specifies which variables to dump into the VCD file. When invoked with no arguments, `$dumpvars` dumps all variables in the design, except those in source-protected regions.

- `$dumpvars (levels [, list_of_modules_or_variables]);`

Specifies which variables to dump into the VCD file. The `levels` argument indicates the number of hierarchical levels below each specified module instance that `$dumpvars` affects. Subsequent arguments specify which scopes of the model to dump to the VCD file. These subsequent arguments can specify entire modules or individual variables within a module.

Examples:

The following invocation dumps all variables within the module `top`; it does not dump variables in any of the modules instantiated by module `top`.

```
$dumpvars (1, top);
```

The following invocation dumps all variables in the module `top` and in all module instances below module `top` in the design hierarchy.

```
$dumpvars (0, top);
```

The following example shows how the `$dumpvars` task can specify both modules and individual variables. 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`.

```
$dumpvars (0, top.mod1, top.mod2.net1);
```

If you want 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 `ncelab -expand` command-line option expands all nets, but this procedure is not recommended due to its negative impact on memory usage and performance.

NC-Verilog 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.

**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`.

■ `$dumpoff;` and `$dumpon`:

These tasks let you specify the simulation period during which the dump takes place.

`$dumpoff` suspends the dump. A checkpoint is created in which every selected variable is dumped as an `x` value. To resume the dump, invoke `$dumpon`.

■ `$dumpall;`

Creates a checkpoint in the dump file that shows the current value of all selected variables.

■ `$dumplimit (filesize);`

Sets a limit on the size of the VCD file. The `filesize` argument specifies the maximum size of the dump file in bytes.

■ `$dumpflush;`

Empties the operating system's dump file buffer to ensure that all the data in that buffer is stored in the dump file. You can also use the PLI `tf_dumpflush()` function in your application program C code.

## Example Source Description Containing VCD Tasks

In the following example, the name of the dump file is `verilog.vcd`. NC-Verilog dumps value changes for all variables in the design. 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.vcd");           // Default file name is verilog.dump

    initial @do_dump
        $dumpvars;                         // Dump variables in the design

    always @do_dump                      // 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
```

## Syntax and Format of the VCD File

A four-state VCD file contains three sections:

- Header information section—Shows the date, the version number of the simulator, and the timescale used in the simulation.
- Node information section—Contains definitions of the scope and type of variables dumped.
- Value changes section—Shows the actual value changes at each simulation time increment. Only variables that change value during a time increment are listed.

See the *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language* (IEEE Std 1364-1995 or 1364-2001) for details on the syntax and format of the VCD file.

## Generating an Extended Value Change Dump (EVCD) File

A value change dump (VCD) file is an ASCII file that contains information about value changes on selected variables in the design. The file contains header information, variable definitions, and the value changes for all specified variables.

The Verilog LRM describes two types of VCD files:

- Four-state—Represents variable changes in 0, 1, x, and z with no strength information.
- Extended—Represents variable changes in all states and strength information.

This section tells you how to generate an extended VCD file. See “[Generating a Value Change Dump \(VCD\) File](#)” on page 458 for information on generating a four-state VCD file.

In NC-Verilog, you can generate an EVCD file by using the `$dumports` system task in your Verilog source description. The implementation of this task in NC-Verilog differs in some ways from the description of the task in the IEEE Std 1364-2001 standard.

NC-Verilog also supports a `$dumports_close` system task. It does not support the other extended VCD system tasks described in the IEEE 1364-2001 standard (`$dumportsoff`, `$dumportson`, `$dumportslimit`, and so on).

You cannot generate an EVCD file for Verilog by using Tcl commands.

### Using the `$dumports` System Task

The `$dumports` system task scans the primary ports of a specified module instance and monitors the ports for both value and drive level. The task generates an output file that contains the value, direction, and strength information for all the ports of a device.

Syntax:

```
$dumports (module_identifier, ["file_pathname"], [ID], [format_flag]);
```

You can specify only one module instance. Use multiple `$dumports` calls to different files if you want to dump data for more than one module.

## NC-Verilog Simulator Help

### Debugging Your Design

---

The following table describes the arguments of `$dumppports`.

Argument	Description
<i>module_identifier</i>	<p>The name of the module instance to be monitored.</p> <p>This argument is required. You cannot specify a top-level module.</p> <p>Paths to modules are allowed, using the period hierarchy separator.</p>
<code>"file.pathname"</code>	<p>A string containing the name of the output file.</p> <p>This argument is optional. If you do not include this argument, the simulator creates a file called <code>verilog.evcd</code> in the current working directory.</p>
<i>ID</i>	<p>An integer data type that identifies a running <code>\$dumppports</code> task with the <code>\$dumppports_close</code> system task. For example,</p> <pre>module ...; ...     integer evcd_id; ... initial begin     \$dumppports(dut1, "dut1.evcd", evcd_id); ... #4000 \$dumppports_close(evcd_id); end ...</pre> <p>This argument is optional.</p> <p>See “<a href="#">Using the <code>\$dumppports_close</code> System Task</a>” on page 465 for more information.</p>

Argument	Description
<i>format_flag</i>	An integer value that determines the value/strength format.  The <i>format_flag</i> argument is optional. It can be one of the following values: <ul style="list-style-type: none"><li>■ 0—Default behavior.</li><li>■ 1—Keep losing value.</li><li>■ 2—Generate output according to the IEEE 1364-2001 standard.</li><li>■ 3—Do both 1 and 2.</li></ul> See " <a href="#">Using the format_flag Argument to Control \$dumpports Output</a> " on page 473 for details on the <i>format_flag</i> argument.

## Examples

The following example generates an output file that contains the value, direction, and strength information for all the ports of `testbench.dut`. The output file is called `verilog.evcd`.

```
$dumpports(testbench.dut);
```

The following example generates an output file called `testoutput.evcd` in the current working directory.

```
$dumpports(testbench.dut, "testoutput.evcd");
```

The following example generates an output file called `testoutput.evcd` in the `./worklib` directory.

```
$dumpports(testbench.dut, "./worklib/testoutput.evcd");
```

The following example generates an EVCD file called `testoutput.evcd`. This `$dumpports` call includes the *format\_flag* argument 1, which specifies that the calculated strengths for both the one and zero components of the value are to be reported.

```
$dumpports(testbench.dut, "testoutput.evcd", , 1);
```

## Using the `$dumpports_close` System Task

The `$dumpports_close` system task stops a running `$dumpports` task. The syntax is as follows:

```
$dumpports_close(ID);
```

The *ID* argument is an integer data type that identifies a particular \$dumpports system task. If only one \$dumpports system task is running, the *ID* argument can be omitted.

## Example

In the following example, two EVCD files are generated: dut1.evcd and dut2.evcd. The dump identified with the ID called id1 is closed after 4000 time units.

```
module top;
    reg A;
    integer id1;
    integer id2;
    ...
    ...
    initial
        begin
            $dumpports(dut1, "dut1.evcd", id1);
            $dumpports(dut2, "dut2.evcd", id2);
            #4000 $dumpports_close(id1);
        end
    ....
    ....
endmodule
```

## Syntax and Format of the EVCD File

The format of the extended VCD file is similar to that of the four-state VCD file, which is described in the IEEE Verilog LRM. The file contains three sections: header information, node information, and value changes.

### Header Information Section

The EVCD file begins with a header in the following format:

```
$date
    <date and time file was generated>
$end
$version
    <version of ncsim>
$end
```

```
$timescale  
    <timescale used for the simulation>  
$end
```

The following is an example header section from an EVCD file:

```
$date  
    Mon Jun 18 13:59:12 2001  
$end  
$version  
    ncsim v03.30.(p001)  
$end  
$timescale  
    1ns  
$end
```

## Node Information Section

This section of the EVCD file begins with the `$scope` keyword command, which defines the scope of the primary ports being dumped, and ends with `$enddefinitions $end`, which marks the end of the header and node information sections.

This section has the following syntax:

```
$scope module <module_instance_name> $end  
$var <var_type>   <port_size>   <port_identifier_code>   <port_reference> $end  
...  
...  
$upscope $end  
$enddefinitions $end
```

The constructs in the lines that begin with the `$var` keyword command are defined as follows:

- *var\_type*—The type of variable. For EVCD, this is always the keyword `port`.
- *port\_size*—The port size is a decimal number indicating the number of bits in the port.
- *port\_identifier\_code*—The identifier for the port. This identifier is used in subsequent message dumping.

**Note:** The signal identifier codes in an EVCD file generated by NC-Verilog differ from what is specified in the IEEE standard. The IEEE standard specifies that the identifier code is to be an integer preceded by `<`, which starts at zero and ascends in one unit increments for each port. For example `<0`, `<1`, and so on. NC-Verilog uses space-efficient identifiers in order to minimize the size of the file. See the example below.

- *port\_reference*—An identifier that indicates the port name.

The following is an example of the node information section in an EVCD file:

```
$scope module board.counter $end
$var port      4 !      value $end
$var port      1 "      clock $end
$var port      1 #      fifteen $end
$var port      1 $      altFifteen $end
$upscope $end

$enddefinitions $end
```

## Value Change Section

The value change section shows the actual value changes at each simulation time increment. Only variables that change value during a time increment are listed. The format of the message is as follows:

```
#<simulation_time>
p<port_value> <0_strength_component> <1_strength_component> <identifier_code>
```

The constructs in the message are defined as follows:

- *#simulation\_time*—The simulation time.
- *p*—Key character that indicates a port.
- *port\_value*—State character that indicates the driving direction and state. The state characters are described in [“Port Value Character Mapping”](#) on page 470.
- *0\_strength\_component*—One of the eight Verilog strengths. This indicates the strength0 component of the value. See [“Strength Mapping”](#) on page 473 for information on strength mapping.
- *1\_strength\_component*—One of the eight Verilog strengths. This indicates the strength1 component of the value.
- *identifier\_code*—The identifier code for the port, which was defined in the *\$var* construct for the port.

The following example shows the node information section and part of the value change section of an EVCD file:

```
$scope module board.counter $end
$var port      4 !      value $end
$var port      1 "      clock $end
$var port      1 #      fifteen $end
$var port      1 $      altFifteen $end
$upscope $end

$enddefinitions $end

#0
pXXXX 6666 6666 !
pN   6 6 "
pX   6 6 #
pX   6 6 $

#5
pU   0 6 "

#10
pLLLL 6666 0000 !
pL   6 0 #
pL   6 0 $

#50
pD   6 0 "

#60
pLLLH 6660 0006 !
...
...
```

## 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;
```

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 giving the scope of the force. 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 output file.

## Port Value Character Mapping

The state information shown in this section is described in terms of input values from a test fixture, the output values of the device under test, and the states that represent unknown direction.

### Direction INPUT

Given a device under test (DUT) and a test fixture, the driving direction is INPUT if the drivers from the test fixture are driving some non-tristated value and the drivers inside the DUT are tristated. The resolved value is mapped as shown in the following table. In the table, the term *active* implies that the drivers are in a non-tristated condition.

**Table 11-1 Driving Direction INPUT Mapping**

D	(0) low
d	(0) low (2 or more drivers active)
U	(1) high
u	(1) high (2 or more drivers active)
N	(X) unknown
n	(X) unknown because of a 1-0 collision
Z	(Z) tri-state

### Direction OUTPUT

If the driving value from drivers inside the DUT is non-tristated, but the value driven by the drivers in the test fixture is tristated, the direction is OUTPUT. The resolved value is mapped as shown in the following table:

**Table 11-2 Driving Direction OUTPUT Mapping**

L	(0) low
I	(0) low (more than 2 drivers active)
H	(1) high
h	(1) high (more than 2 drivers active)
X	(X) unknown (don't care)
T	(Z) tri-state

## Direction UNKNOWN

If both the drivers in the test fixture and drivers inside the DUT are driving some non-tristated value, the direction is UNKNOWN. The resolved value is mapped as shown in the following table:

**Table 11-3 Driving Direction UNKNOWN Mapping**

---

0	(0) low (both input and output are active with 0 value)
1	(1) high (both input and output are active with 1 value)
?	(X) unknown (input X and output X)
F	tri-state (input and output unconnected)
A	(0-1) unknown (input 0 and output 1)
a	(0-X) unknown (input 0 and output X)
B	(1-0) unknown (input 1 and output 0)
b	(1-X) unknown (input 1 and output X)
C	(X-0) unknown (input X and output 0)
c	(X-1) unknown (input X and output 1)
f	(Z) unknown (input and output tri-stated)

---

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 `$dumports` output are shown in the following table. Append 0 or 1 to the keyword as appropriate for the strength component.

0	highz (HiZ)
1	small (Sm)
2	medium (Me)
3	weak (We)
4	large (La)
5	pull (Pu)
6	strong (St)
7	supply (Su)

## Using the `format_flag` Argument to Control `$dumports` Output

You can control the output of `$dumports` with the `format_flag` argument. This argument can be one of the following values:

- 0—Default behavior.

Report the strengths for both the zero and one components of the value if the strengths are the same. If the strengths are different, report only the “winning” strength. That is, the two strength values either match (for example, `pA 5 5 !`) or the winning strength is shown and the other is zero (for example, `pH 0 5 !`).

- 1—Keep losing value.

Report the strengths for both the zero and one components of the value (for example, `pD 6 5 !`).

■ 2—Generate output according to the IEEE 1364-2001 standard.

The IEEE standard states that the values 0 (both input and output are active with value 0) and 1 (both input and output are active with value 1) are conflict states. The standard then defines two strength ranges:

- ❑ Strong: strengths 7, 6, and 5
- ❑ Weak: strengths 4, 3, 2, 1

The rules for resolving conflicts are:

- ❑ If the input and output are driving with the same range of strength, the resolved value is 0 or 1, and the strength is the stronger of the two.
- ❑ If the input is driving a strong strength and the output is driving a weak strength, the resolved value is d or u, and the strength is the strength of the input.
- ❑ If the input is driving a weak strength and the output is driving a strong strength, the resolved value is l or h, and the strength is the strength of the output.

■ 3—Do both 1 and 2.

### Example

In this example, the *format\_flag* argument to the \$dumpports call was set to each of its four possible values to generate four EVCD files. The source file is shown below. The table that follows the source file shows the value change section for the four output files. Notice that, if the *format\_flag* argument is set to 1 or to 3, strengths for both the zero and one components of the value are reported.

```
'timescale 1 ns / 1 ns
module test_bench;
    reg AS_3, AS_2, AS_1;
    tri TF;

    // The following 3 continuous assignments are the drivers from the test fixture
    assign (strong1, strong0) TF = AS_1; // will make TF have strength 6
    assign (pull1, pull0)      TF = AS_2; // will make TF have strength 5
    assign (weak1, weak0)      TF = AS_3; // will make TF have strength 3

    initial
        begin
            $dumpports(DUT, "test.evcd", ,0);
            assign AS_3 = 1'bz;
            assign AS_1 = 1'bz;
```

## NC-Verilog Simulator Help

### Debugging Your Design

---

```
assign AS_2 = 1'bz;
#10;

assign AS_3 = 1'bz;
assign AS_2 = 1'bz;
assign AS_1 = 0;
#10;

assign AS_3 = 0;
assign AS_1 = 1'bz;
assign AS_2 = 1'bz;
#10;
end

buf_w_strength DUT(TF);
endmodule

module buf_w_strength (IO);
 inout IO;

 wire IO_wire;
 pullup (pull1, pull0) (IO_wire); // This is the driver within the DUT
 // Assignment will make IO have a strength of 5
 assign (pull1, pull0) IO = IO_wire;
endmodule
```

**Table 11-4 Effect of format\_flag Argument**

<b>format_flag Argument</b>			
<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>
#0	#0	#0	#0
pH 0 5 !	pH 0 5 !	p1 0 5 !	p1 0 5 !
#10	#10	#10	#10
pD 6 0 !	pD 6 5 !	pA 6 0 !	pA 6 5 !
#20	#20	#20	#20
pH 0 5 !	pH 3 5 !	pA 0 5 !	pA 3 5 !

## \$dumports Restrictions

The following restrictions apply to the \$dumports system task:

- The \$dumports system task does not work with save and restart.
- 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 u1(out, 1'b1, 1'b1);
DUT uudut(out);
...

module DUT(out)
  tran t1(out, int);

```

## Comparing Databases with Comparescan

Use Comparescan to compare simulation histories stored in SHM (SST2) or VCD databases.

Comparescan is a comprehensive comparison tool that can compare single signals or complete simulations. Using these comparisons, you can verify that different simulation runs produce functionally equivalent results.

**Note:** Comparescan is not available on Windows platforms. However, the SST2 database is platform independent, so you can use Comparescan on UNIX to compare databases created on Windows.

You can use Comparescan to compare simulations performed:

- At the same or at different levels (RTL to RTL, gate to gate, RTL to gate, behavioral to gate, and so on).
- After design optimizations (for speed or area).
- After technology changes (shrink or vendor).
- Using different clock rates.
- For regression testing.

For test vector applications, you can use Comparescan to perform comparisons:

- Of best and worst case timing simulations.
- Of simulations before and after backannotation.
- After clock tree insertions.

You can also use Comparescan to compare simulations performed on different simulators. For example, you can compare simulation results from Verilog-XL and results from the NC-Verilog simulator.

You cannot compare a VCD database with an SHM database.

See the [Comparescan User Guide](#) for details on using Comparescan.

## Generating a Code Coverage Database File

Code coverage is a verification tool that can increase your design and verification productivity by identifying and quantifying the parts of your design that have and have not been exercised during simulation. By providing detailed coverage information, code coverage quickly identifies the untested areas of your design and any weaknesses in your test suite. You can then create additional tests to target those areas of the design. You can also use code coverage to identify redundant tests that you can remove from a regression suite or set of production test vectors. This helps you to reduce your test generation time without sacrificing design quality.

Code coverage is fully integrated with the NC-Verilog, NC-VHDL, and NC-Sim simulators.

**Note:** Code coverage is not currently supported on Windows platforms.

The code coverage tool consists of two components: the code coverage recorder, which generates a database that contains information on those parts of the design that were exercised during simulation, and the coverage analyzer, which lets you examine the coverage results. You can analyze your coverage results interactively, generate batch reports, and merge coverage results databases.

Three types of coverage are supported:

- Statement coverage

Statement coverage reports whether a statement has executed or has not executed during a given simulation run. This verifies that the simulation has covered all of the statements in your source code and identifies areas of dead code. The code coverage analyzer reports the statement coverage statistic in boolean format. Boolean format reports 1 if the statement has executed and 0 if the statement has not executed.

- State machine (fsm) coverage

State machine (fsm) coverage tells you which states the state machine variables have been in (state visitation), and what state-to-state transitions have taken place (state transitions).

- Expression coverage

Expression coverage identifies which terms of an expression in your code contribute to true and false values for the expression's terms during a simulation run.

**Note:** Using the expression recording features can severely impact your simulation performance.

The code coverage tool records design information in a design database directory named `cover.cov` by default. The design database directory contains the following database files:

- The design database file (`.dsn`) contains information about the design. The code coverage tool updates the design database file only if the design has changed since the last simulation run. This practice lets you use multiple testbenches to exercise the same design and merge the coverage results during analysis.
- The code coverage database file (`.acv`) contains the coverage metrics, or results, for your simulation run, which you can then analyze with the coverage analyzer. You can optionally specify an analysis control file that lets you control the analysis using commands that correspond to the coverage functions available in the coverage analyzer.

The database files are given version numbers. If database files exist in a specified directory and you create new database files with identical names, the code coverage tool increments the version numbers of the new files.

To record code coverage data, you:

1. Compile the Verilog source files with `ncvlog`, and/or compile the VHDL source files with `ncvhdl`.

For expression coverage, the Verilog modules (and VHDL design units) must be compiled with the `-linedebug` option.

2. Elaborate the design with the `ncelab -coverage` option (`ncverilog +nccoverage`).

For state machine coverage, you must specify read access to simulation objects with the `ncelab -access +r` option (`ncverilog +ncaccess+r`).

3. If you are using the command-line interface, invoke the simulator and then use the Tcl `coverage` command to specify the type of coverage that you want and to control the dumping of code coverage data. See the *Code Coverage User Guide* for details on the `coverage` command.

If you are using the SimVision analysis environment, invoke the simulator with the `-gui` option, and then use the commands on the *Coverage* menu. See “[Gathering Code Coverage Data](#)” in the *SimVision Analysis Environment User Guide* for details on using the *Coverage* menu commands.

## Displaying Debug Settings

While debugging, you may open databases, set probes, set breakpoints, set aliases, and so on. To display your current debug settings:

- If you are using the Tcl command-line interface, use the appropriate modifier to display information. For most commands, this is the `-show` modifier. For example:

```
ncsim> database -show  
ncsim> probe -show
```

Use the `alias` command without a modifier to display information about aliases you have set, as shown in the following example:

```
ncsim> alias  
f2      finish 2  
h       history  
ncsim>
```

- If you are using the SimVision analysis environment, use the commands on the *Show* menu. For example, to view information on the breakpoints you have set, select *Show—Breakpoints*.

See “[Displaying Debug Settings](#)” in the *SimVision Analysis Environment User Guide* for an example.

## Setting a Default Radix

If you are using the SimVision analysis environment, select *Options—Default Radix* to select a default radix for the simulator to use in displaying signal values.

The radix you choose is used to display values when you:

- Double click on an object in the Source Browser.
- Position your cursor over an object in the Source Browser.
- Right click on an object in the Source Browser and select *Show Value* from the pop up.
- Click on the *Show Value* button on the SimControl tool bar.



- Open a new Watch Window.
- Select *Set—Force* to open the Set Force form.

The Show Value form has its own buttons so that you can override the default radix for particular operations.

## Setting Variables

You can set Tcl variables to help you debug your design. In addition to user-defined variables, the simulator includes several predefined Tcl variables that you can use to control various simulator features.

You can:

- Set a variable or change the value of a variable with the built-in `Tcl set` command.

```
ncsim> set abc 10  
ncsim> set vlog_format %b  
ncsim> set pack_assert_off {std_logic_arith}
```

- Delete a variable, with the `unset` command.

```
ncsim> unset abc
```

- Display a list of predefined simulation variables and their current values, with the `help -variables` command.

```
ncsim> help -variables
```

- Display a list of all currently set variables, with the `info vars` command. This command does not display variable values.

```
ncsim> info vars
```

You can put variable definitions in an input file and then execute the commands in this file by using the `-input` option when you invoke the simulator. You can also execute these commands by using the `source` or `input` command or the *File—Input Commands* menu command after invoking the simulator.

See “[Setting Variables](#)” in the *SimVision Analysis Environment User Guide* for an example of using SimControl to set a Tcl variable, display a list of variables that have been set, change the value of a predefined variable, and unset a variable.

The predefined Tcl variables are:

**`assert_1164_warnings = value`**

Controls the display of warnings from builtin functions from the std\_logic\_1164 package. The `value` can be `yes` or `no`. If you set it to `no`, the simulator suppresses the warnings. This variable is initially set to `yes`.

**`assert_report_level = value`**

Sets the minimum severity level for which VHDL assertion report messages should be output. The `value` can be `note`, `warning`, `error`, `failure`, or `never`.

If the severity level specified in the assertion statement is at or above the severity level specified by this variable, the report message is written to standard output, along with the time, severity, and the name of the design unit in which the assertion occurred.

This variable is initially set to `note`.

**`assert_stop_level = value`**

Specifies the minimum severity level for which VHDL assertions cause the simulation to stop.

The `value` can be `note`, `warning`, `error`, `failure`, or `never`.

If the severity level specified in the assertion statement is at or above the severity level specified by this variable, the simulation stops. If the simulator is in interactive mode, it returns the Tcl prompt. If not in interactive mode, the simulator exits.

This variable is initially set to `error`.

**`autoscope = value`**

The value of this variable can be `yes` or `no`.

- `yes`—Set the debug scope to the scope of the current execution point (if any) when the simulator stops.
- `no`—Do not automatically set the debug scope to the scope of the current execution point (if any) when the simulator stops.

This variable is initially set to `yes`.

**clean = value**

The value of this variable can alternate between 1 and 0 as the simulation runs:

- 1—The simulation is in a clean state, and you can use the `save -simulation` command to create a snapshot of the current simulation state.
- 0—The simulation is not in a clean state. If you want to save the simulation state, use the `run -clean` command to run the simulation to the next point at which the `save` command will work.

**display\_unit = value**

The value of this variable is the time unit used to display time values throughout the user interface.

- `auto`—Use the largest time unit in which the time can be expressed as an integer.
- `module`—Use the timescale of the module that is the current debug scope.
- `FS, 10FS, 100FS, ...`
- `xlstyle`—Print time values using the same formatting rules that Verilog-XL uses. XL follows any `$timeformat` that is in effect, and, if there is none, formats time values to the smallest ``timescale` precision. Setting `display_unit` to `xlstyle` can make it easier to compare simulation results from the two simulators.

Setting the value to `xlstyle` affects the formatting of time values only. It does not affect the format of messages.

The default value is `auto` unless you have invoked the simulator (`ncsim`) with the `-xlstyle_units` command-line option, in which case the default value is `xlstyle`.

**pack\_assert\_off = value**

The value of this variable specifies the package name(s) from which you want to suppress the display of VHDL assert messages. The value is:

```
{ [library.]package [[library.]package ...] }
```

If you specify more than one package, use a space (not a comma) to separate the package names. For example,

```
ncsim> set pack_assert_off {std_logic_arith numeric_std}
```

You can specify a library name if there are packages with the same name in different libraries and you want to turn off messages only in a package in a particular library. If you don't specify a library name, assert messages are turned off in all packages with the specified name.

Set the `severity_pack_assert_off` variable to specify the severity level(s) of the messages that you want to suppress.

**`severity_pack_assert_off = value`**

The value of this variable specifies the severity level of the VHDL assert messages that originate from IEEE or user-defined packages that you want to suppress in the simulation run. The value can be one or more of the following:

- failure
- error
- warning
- note

The value of this variable is initially set to {warning note}.

If you specify more than one severity level, separate the levels with a space. For example,

```
ncsim> set severity_pack_assert_off {error warning note}
```

After you have specified the severity level of the assert messages that you want to suppress, you must specify the package names by setting the `pack_assert_off` variable.

**`snapshot = value`**

The value of this variable is the name of the currently loaded simulation snapshot. This variable is read-only.

**`tcl_prompt1 = value`**

The value of this variable is the command that generates the main Tcl prompt. The default is:

```
puts -nonewline "ncsim> "
```

The following command changes the prompt to ncverilog>:

```
ncsim> set tcl_prompt1 {puts -nonewline "ncverilog> "}
```

**tcl\_prompt2 = value**

The value of this variable is the command that generates the prompt you get if you press the Return key before completing a Tcl command. The default is:

```
puts -nonewline "> "
```

The following command changes the prompt to Give me more>:

```
ncsim> set tcl_prompt2 {puts -nonewline "Give me more> "}
```

**time\_unit = value**

The value of this variable is the unit to be used in time or cycle specifications that do not contain an explicit unit. The value can be: FS, 10FS, 100FS, ... , SEC, MIN, HR, CYCLE, or module. For example, if time\_unit = NS, the following command runs the simulation for 100 ns.

```
ncsim> run 100
```

This variable is initially set to module, which uses the timescale of the module that is the current debug scope.

**time\_scale = value**

The value of this variable is the timescale of the current debug scope. This variable is read-only.

**vhdl\_format = value**

The value of this variable is the format for the output of VHDL values in describe output, stop point messages, and expression results. The value can be set to %h, %x, %d, %o, or %b, or %v. This variable is initially set to %v.

**vlog\_format = value**

The value of this variable is the format for the output of Verilog values in describe output, stop point messages, and expression results. The value can be set to %h, %x, %d, %o, or %b.

This variable is initially set to %h.

## Suppressing Assert Messages in IEEE or User-Defined Packages

When simulating a VHDL design, you may want to turn off assert messages that come from IEEE or user-defined packages, especially assert messages with a severity level of warning and note, and assert warning messages from built-in operators. You can do this by using the Tcl `set` command to set two variables: `severity_pack_assert_off` and `pack_assert_off`.

1. Specify the severity level of the messages that you want to suppress by setting the `severity_pack_assert_off` variable. The syntax is as follows:

```
set severity_pack_assert_off {severity_level}
```

The `severity_level` value can be one or more of the following:

- failure
- error
- warning
- note

If you specify more than one severity level, separate the levels with a space.

### Examples

```
ncsim> set severity_pack_assert_off {note}
ncsim> set severity_pack_assert_off {warning}
ncsim> set severity_pack_assert_off {error warning note}
```

The value of `severity_pack_assert_off` is initially set to `{warning note}`.

**Note:** You cannot turn off different types of messages for different packages.

2. Specify the package name(s) from which you want to suppress the display of VHDL assert messages by setting the `pack_assert_off` variable. The syntax is as follows:

```
set pack_assert_off {package_specification}
```

The `package_specification` value is:

```
[library.]package [[library.]package ...]
```

If you specify more than one package, use a space (not a comma) to separate the package names.

You can specify a library name if there are packages with the same name in different libraries and you want to turn off messages only in a package in a particular library. If you

don't specify a library name, assert messages are turned off in all packages with the specified name.

### Examples

```
ncsim> set pack_assert_off {numeric_std}  
ncsim> set pack_assert_off {std_logic_arith numeric_std}
```

If you are using the SimVision Analysis Environment, you can set these variables by selecting *Show—Tcl Variables*. On the Debug Settings form, select the variable and then enter the value in the *Value* field at the bottom of the form. See “[Setting Variables](#)” in the *SimVision Analysis Environment User Guide* for an example of using SimControl to set a Tcl variable or to change the value of predefined variables.

The NC VHDL simulator also has a predefined Tcl variable called `assert_1164_warnings`. This variable controls the display of warnings from built-in functions from the `std_logic_1164` package. This value of this variable is initially set to `yes`, which tells the simulator not to suppress these warnings. If you set the value to `no`, the simulator suppresses the warnings from this particular package.

Some packages include other packages. To turn off assertions that come from functions in an included package, you must specify the included package. To determine what package(s) the messages that you want to suppress come from, look at the actual assert messages generated in the `ncsim.log` file.

The following UNIX command is useful for determining which packages to filter:

```
% grep ieee ncsim.log | cut -d'@' -f2 | cut -d',' -f1 | sort | uniq
```

Examples:

The following commands turn off assert warning messages from package `numeric_std`. The first command is optional if the `severity_pack_assert_off` variable is already set to `warning`.

```
ncsim> set severity_pack_assert_off {warning}  
ncsim> set pack_assert_off {numeric_std}
```

The following commands turn off assert warning messages from packages `numeric_std` and `std_logic_arith`.

```
ncsim> set severity_pack_assert_off {warning}  
ncsim> set pack_assert_off {numeric_std std_logic_arith}
```

The following commands turn off assert warning messages from packages `numeric_std`, `std_logic_arith`, and `mytypes`, which is in the library `mypad`.

```
ncsim> set severity_pack_assert_off {warning}  
ncsim> set pack_assert_off {numeric_std std_logic_arith mypack.mytypes}
```

## Editing a Source File

If you are using the SimVision analysis environment, you can open a source file for editing in your editor from the SimControl window. To specify an editor, select *Options—Preferences*. On the Preferences form, specify the editor in the “Editing” section of this form. See “[Editing](#)” in the *SimVision Analysis Environment User Guide* for an example.

You can invoke your editor in two ways:

1. Click on the *Edit Source* button on the SimControl tool bar.



This invokes your editor on the file currently displayed in the Source Browser. The cursor is at the first line visible in the Source Browser.

2. Select *File—Edit File*.

The Edit File form appears. Use this form to specify the filename and the line number to bring up in the editor.

- In the *Filename* field, enter the name of the file you want to edit.
- In the *Line Number* field, enter the line number you want to edit.
- Click *OK* or *Apply* to invoke your editor.

See “[Editing a Source File](#)” in the *SimVision Analysis Environment User Guide* for an example.

## Searching for a Line Number in the Source Code

If you are using the SimVision analysis environment, the *File—Find—Line* command lets you quickly find a line of text in the source code displayed in the Source Browser.

See “[Searching for a Line Number in the Source Code](#)” in the *SimVision Analysis Environment User Guide* for an example.

## Searching for a Text String in the Source Code

If you are using the SimVision analysis environment, the *File—Find—Text* command lets you quickly locate a text string in the source code displayed in the Source Browser.

See “[Searching for Text in the Source Code](#)” in the *SimVision Analysis Environment User Guide* for an example.

## Configuring Your Simulation Environment

When you use the SimVision analysis environment, you can configure your simulation environment to reflect your preferences. To set your preferences, select *Options—Preferences*. This command lets you:

- Define the behavior of warning pop-ups.
- Control the behavior of the Source Browser.
- Specify a text editor.
- Select the size of the display fonts.

See “[Configuring Your Simulation Environment](#)” in the *SimVision Analysis Environment User Guide* for an example of using the Preferences form.

## Saving and Restoring Your Simulation Environment

You can save the current state of the debug environment at any time.

- If you are using the Tcl command-line interface, use the `save -environment` command.

```
ncsim> save -environment [filename]
```

This command generates a script containing Tcl commands to recreate breakpoints, databases, probes, and the values of Tcl variables. The *filename* argument is optional. If not specified, the script is written to standard output.

See “[save](#)” on page 602 for more information on the `save -environment` command and for an example.

To restore the environment, execute the script with the Tcl `source` command or use the `-input` option when you invoke the simulator.

- If you are using the SimVision analysis environment, select *File—Debug Environment—Save* to create the script. Select *File—Debug Environment—Restore* to restore your debug settings.

See “[Saving and Restoring Your Debug Environment](#)” in the *SimVision Analysis Environment User Guide* for an example.

When you source a script containing Tcl commands to restore a saved debug environment or use the *File—Debug Environment—Restore* command, the debug settings in the script are merged with your current debug settings.

**Note:** These scripts are meant to be sourced into a “clean” environment. That is, typically you source the script (or use *File—Debug Environment—Restore*) after you have invoked the simulator, but before you set any breakpoints or probes or open a database.

If you invoke the simulator, set some breakpoints and probes, and then source a script that contains commands to set breakpoints and probes, the simulator will probably generate errors telling you that some commands in the script could not be executed. These errors are due to name conflicts. For example, you may have set a breakpoint that received the default name “1”, and the command in the script is trying to create a breakpoint with the same name. You can, of course, give your breakpoints unique names to avoid this problem. You can also edit the scripts to make them work the way you would like them to work.

## Creating or Deleting an Alias

You can create your own shorthand for a command or series of commands by setting an alias.

- If you are using the Tcl command-line interface, use the `alias` command to create a new alias.  
See “[alias](#)” on page 498 for details on using the `alias` command and for an example.
- If you are using the SimVision analysis environment, select *Set—Alias* and fill in the Set Alias form with the name and definition of the alias.  
To display the aliases that are currently set, select *Show—Aliases*.  
See “[Creating or Deleting an Alias](#)” in the *SimVision Analysis Environment User Guide* for an example.

You cannot create an alias with the same name as a predefined Tcl command. For example,

```
ncsim> alias gets value  
ncsim: *E,ALNORP: cannot create an alias for Tcl command gets.
```

## Getting a History of Commands

You can get a listing of all the commands you have entered by using the built-in Tcl `history` command. The `history` command lets you re-execute commands without retyping them. You also can use the `history` command to modify old commands—for example, to fix typographical errors.

- If you are using the Tcl command-line interface, enter the `history` command at the `ncsim>` prompt.  
See “[history](#)” on page 557 for details on the `history` command and for an example.
- If you are using the SimVision analysis environment, select *Show—History*. The list of commands is displayed in the I/O region of the window.  
See “[Getting a History of Commands](#)” in the *SimVision Analysis Environment User Guide* for an example.

## Managing Custom Buttons

If you are using the SimVision analysis environment, you can create custom buttons for commands and add them to the Tool Bar. You can assign one or multiple Tcl commands to a custom button. Customized buttons can automate your debugging tasks by letting you execute a series of Tcl commands with one mouse click. You save time by avoiding having to select individual Tcl commands for performing frequent tasks.

You can:

- Create buttons that perform a specified function.
- Edit the buttons to change the functions they perform.
- Reorder the buttons on the Tool Bar.
- Export the buttons to save them in a file.
- Import a file of user-defined buttons.

See “[Managing Custom Buttons](#)” in the *SimVision Analysis Environment User Guide* for details and examples.

---

# Using the Tcl Command-Line Interface

---

## Overview

The following simulator commands are available to help you debug your design:

<u>alias</u>	<u>exit</u>	<u>probe</u>	<u>stack</u>
<u>attribute</u>	<u>finish</u>	<u>process</u>	<u>status</u>
<u>call</u>	<u>fmibkpt</u>	<u>release</u>	<u>stop</u>
<u>check</u>	<u>force</u>	<u>reset</u>	<u>strobe</u>
<u>coverage</u>	<u>help</u>	<u>restart</u>	<u>task</u>
<u>database</u>	<u>history</u>	<u>run</u>	<u>time</u>
<u>deposit</u>	<u>input</u>	<u>save</u>	<u>value</u>
<u>describe</u>	<u>memory</u>	<u>scope</u>	<u>version</u>
<u>drivers</u>	<u>omi</u>	<u>source</u>	<u>where</u>

The simulator command language is based on Tcl. The language is object-oriented, which means that the action to be performed on the object is supplied as a modifier to the command. For example, in the following command, `database` is the object, and `-open` is a modifier.

```
ncsim> database -open
```

The command format is:

```
ncsim> command [-modifier] [-options] [arguments]
```

- Commands consist of a command name, which may be followed by either *arguments* or *-modifiers*. The command name is always the first or left-most word in the command. *-modifiers* may have *-options*.
- Commands can be entered in upper or lowercase.
- Commands can be abbreviated to the shortest unique string.

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

Here are some example Tcl commands:

```
ncsim> alias           (command)
ncsim> alias myalias    (command, argument)
ncsim> probe -show      (command, modifier)
ncsim> probe -show myprobe (command, modifier, argument)
ncsim> probe -create -all (command, modifier, option)
ncsim> probe -create -all -name myprobe
                  (command, modifier, option, argument)
```

You can enter more than one command on the command line. Use a semicolon to separate the commands.

Because of the way that Tcl works, only the output from the last command in a script or on the command line is printed to the screen or to the logfile. In the following example, only the output of the help command is printed.

```
ncsim> status; help
```

The last section in this chapter, [“Verilog-XL and NC-Verilog Simulator Interactive Debug Commands”](#) on page 665, contains tables listing Verilog-XL commands and their Tcl equivalents.

In addition to the simulator commands listed above, you can also use any Tcl built-in command. See [Appendix A, “Basics of Tcl,”](#) for information on Tcl syntax and on the extensions that have been added to the Tcl interpreter.

## Executing UNIX Commands

You can also execute UNIX commands from within the simulator by entering them at the `ncsim>` prompt. You can do this whether you are using the command-line interface or the SimVision analysis environment. Command output goes to the log file. If you are using the analysis environment, output goes to the I/O region of the SimControl window. For example:

```
ncsim> ls
ncsim> pwd
ncsim> which ncelab
```

You can run UNIX commands in the background by including the ampersand character.

```
ncsim> xterm &
```

You also can use the Tcl `exec` command to run UNIX commands.

```
ncsim> exec xterm &
```

## Using Wildcards Characters in Tcl Commands

You can use wildcard characters in arguments to some Tcl commands. The wildcard characters are:

- The asterisk (\*)

This character stands for any combination of zero or more characters. For example, the pattern `s*n` matches any object name, regardless of length, that starts with the letter `s` and that ends with the letter `n`. Possible matches include `sn`, `sun`, `son`, and `sudden`.

- The question mark (?)

This character stands for any single character. For example, the pattern `p?n` matches any three character object name that starts with the letter `p` and that ends with the letter `n`. Possible matches include `pun`, `pan`, and `pin`.

You can use wildcard characters in two types of arguments:

- In the names of objects that you create with commands such as `probe -create` or `stop -create`.

When you create a probe or set a breakpoint, the simulator gives the probe or breakpoint either a default name or a name that you specify with the `-name` option. You can use wildcard characters to delete, disable, or enable these probes or breakpoints. For example:

```
ncsim> stop -delete *      ;# Deletes all breakpoints

ncsim> stop -disable br*    ;# Disables all breakpoints that have names that
                           begin with br.

ncsim> stop -enable break?  ;# Enables all breakpoints that have names that
                           begin with break and that have one additional
                           character.
```

- In the names of Verilog and VHDL objects in Tcl commands that can take multiple object names as their arguments. When used by itself, `*` matches every object name in the specified scope. If the scope is not specified, it matches every object in the current scope. For example:

```
ncsim> stop -create -object *  ;# Creates a breakpoint on all objects in the
                               current scope.

ncsim> describe mypin*z      ;# Describes all objects in the current scope
                               that have names that start with mypin and
                               that end with z.
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
ncsim> probe -create -shm :dut2.tes* ;# Creates a probe on all objects that have names that begin with dut2.tes.
```

```
ncsim> value :process1:pin?z ;# Displays the values of objects that have names that have five characters and that begin with pin and end with z in the scope :process1.
```

You cannot use wildcard characters in scope specifiers or inside escaped names. For example, the following commands are not valid:

```
ncsim> probe -create -shm top.u*.sig1
```

```
ncsim> stop -create -object :\_\_sig*\_\_ ;# The wildcard character will be considered to be part of the escaped name.
```

When wildcards are used in an object name, signal attributes or array subscripts are not allowed. For example, the following commands are not valid:

```
ncsim> value ab*c'delayed
```

```
ncsim> value ab*c[4]
```

## Command Description Conventions

The following conventions are used in the command reference section:

■ **literal**

Nonitalic words indicate keywords that you must enter literally. These keywords represent command, modifier, or option names.

In the following examples, `run` and `-step` are command and modifier names, respectively.

```
ncsim> run
```

```
ncsim> run -step
```

■ **argument**

Words in italics indicate user-defined arguments for which you must substitute a name or a value. Arguments are case sensitive if used in directory paths or if they refer to Verilog objects.

In the following command, you must enter the name of an object for the `object_name` argument.

```
ncsim> value object_name
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

- [ ]

Square brackets denote optional arguments.

In the following example, the *probe\_name* argument is optional.

```
ncsim> probe -show [probe_name]
```

- |

Vertical bars (OR-bars) separate possible choices for a single argument.

- { }

Curly braces are used with OR-bars and enclose a list of choices from which you must choose one.

For example, the following syntax indicates that one of the three keywords (name, kind, or declaration) *must* be specified:

```
ncsim> scope -describe -sort {name | kind | declaration}
```

- ...

Three dots ( ...) indicate that you can repeat the previous argument. If they are used with brackets, you can specify zero or more arguments. If they are used without brackets, you need to specify at least one argument, but you can specify more. For example:

*argument* ...                    (Specify at least one, but more are possible)

[*argument* ...]                (You can specify zero or more)

## alias

The `alias` command lets you define aliases that you use as command short-cuts. You can:

- Define an alias (using the optional `-set` modifier).
- Display the definition of a specific alias using the alias name as an argument. If the argument is omitted, all aliases and their definitions are displayed.
- Remove an alias definition (`-unset`).

You cannot create an alias with the same name as a predefined Tcl command. For example, the `puts` command is a Tcl command, so you cannot do the following:

```
ncsim> alias puts value
ncsim: *E_ALNORP: cannot create an alias for Tcl command puts.
```

### alias Command Syntax

```
alias [-set] alias_name alias_definition

        -unset alias_name

        [alias_name]
```

### alias Command Modifiers and Options

The `alias` command has two modifiers: `-set`, which creates an alias, and `-unset`, which deletes an alias definition.

The `alias` command, with no modifiers or arguments, prints all alias definitions. If you want to print the definition of a particular alias, include the alias name on the command line.

#### **`-set alias_name alias_definition`**

Creates a command alias. The `-set` modifier is optional.

#### **`-unset alias_name`**

Removes an alias definition. You must include the `alias_name` argument to specify the name of the alias that you want to remove. This is equivalent to the UNIX `unalias` command.

## alias Command Examples

The following command creates an alias called `h`, which is defined as the `history` command. The `-set` modifier is optional.

```
ncsim> alias -set h history
```

The following command creates an alias called `bp` and defines it as the `stop -show` command.

```
ncsim> alias bp stop -show
```

The following command creates an alias called `go` and defines it as `{run 10;value count}`.

```
ncsim> alias go {run 10;value count}
```

The following command prints all alias definitions.

```
ncsim> alias
bp      stop -show
go      run 10;value count
h       history
```

The following command prints the definition of the alias `bp`.

```
ncsim> alias bp
bp      stop -show
```

The following command deletes the alias `bp`.

```
ncsim> alias -unset bp
```

In the following command, the `go` alias is used to advance the simulation and show the value of `count`.

```
ncsim> go
      5count= x
4'hx
```

In the following command, the `h` alias is used as a shortcut for the `history` command.

```
ncsim> h
1  alias h history
2  alias bp stop -show
3  alias go {run 10;value count}
4  alias
5  alias bp
6  go
7  h
```

## attribute

The `attribute` command enables VHDL function-valued attributes for specified signals so that they can then be accessed from the Tcl interface with the `value` command.

The function-valued signal attributes that are supported are:

- ‘EVENT
- ‘LAST\_EVENT
- ‘ACTIVE
- ‘LAST\_ACTIVE
- ‘LAST\_VALUE

See the VHDL LRM for a complete description of these attributes.

If you use the `attribute` command to enable attributes on a signal, all five function-valued attributes are automatically enabled on the specified signal.

Function-valued attributes that you use in the design are automatically enabled on the prefix signal. All five attributes are automatically enabled on the prefix signal. For example, if you use `clk’EVENT` in the design, all function-valued attributes are enabled for `clk`.

If you issue the `attribute` command after simulation has begun, the attributes assume the default values that they would have had at the start of the simulation. The attribute values will be accurate only after the values have been updated during the simulation.

Once you have enabled function-valued attributes for a signal, you cannot disable them.

You cannot use the `attribute` command to enable signal-valued attributes (‘DELAYED, ‘STABLE, ‘QUIET, ‘TRANSACTION). These signal attributes are enabled only if they are used in the design.

See “[Accessing Signal Attributes Using Tcl Commands](#)” in the *NC-Verilog Simulator Help* for more information.

## attribute Command Syntax

```
attribute signal_name [signal_name ...]
```

## **attribute Command Options and Modifiers**

None.

## **attribute Command Examples**

The following VHDL source code is used for the examples in this section.

```
library ieee;
use ieee.std_logic_1164.all;

entity d_flop is
    generic (setup_time, hold_time : time );
    port (d, clk : in std_logic;
          q : out std_logic);

begin
    setup_check : process (clk)
    begin
        if (clk = '1') and (clk'event) then
            assert (d'last_event <= setup_time)
                report "setup violation"
                severity error;
        end if;
    end process setup_check;

    hold_check : process (clk'delayed(hold_time))
    begin
        if (clk'delayed(hold_time) = '1') and
            (clk'delayed(hold_time)'event) then
            assert (d'last_event = 0 ns) or
                (d'last_event < hold_time)
            report "hold violation"
            severity error;
        end if;
    end process hold_check;
end d_flop;
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
architecture d_flop_behave of d_flop is
begin
    dff_process : process (clk)
    begin
        if (clk = '1') and (clk'event) then
            q <= d;
        end if;
    end process dff_process;
end d_flop_behave;
```

```
% ncvhdl -nocopyright d_flop.vhd
% ncelab -nocopyright -generic "setup_time => 10 fs"
    -generic "hold_time => 10 fs" WORKLIB.D_FLOP:D_FLOP_BEHAVE
% ncsim -tcl WORKLIB.D_FLOP:D_FLOP_BEHAVE
ncsim: v3.10.(p1): (c) Copyright 1995 - 2000 Cadence Design Systems, Inc.
Loading snapshot worklib.d_flop:d_flop_behave ..... Done
ncsim> run 400
Ran until 400 FS + 0
```

In the code example shown above, `clk'event` and `d'last_event` are used in the design. Therefore, all function-valued attributes are automatically enabled for the signals `clk` and `d`.

```
ncsim> value clk'event
FALSE
ncsim> value clk'last_event
400 FS
ncsim> value clk'last_value
'U'
ncsim> value d'last_event
400 FS
ncsim> value d'last_value
'U'
```

In the code example, no attributes are used on the `q` signal. Use the `attribute` command to enable function-valued attributes for this signal. The attributes assume the default value that they would have had at the start of the simulation. The accurate value of the attributes is only available after the simulation has been advanced and the values have been updated.

```
ncsim> attribute q
ncsim> run 400
Ran until 800 FS + 0
ncsim> value q'event
FALSE
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
ncsim> value q'last_active  
800 FS
```

Signal-valued attributes ('DELAYED, 'STABLE, 'QUIET, 'TRANSACTION) are enabled only if you have used them in the design. You cannot enable these attributes by using the attribute command.

```
ncsim> attribute q  
ncsim> value q'quiet  
ncsim: *E,BASGAT: Unavailable attribute name.
```

## call

The `call` command lets you call a user-defined C-interface function or a Verilog user-defined PLI system task or function from the command line.

### call Command Syntax

```
call [-systf | -predefined] task_or_function_name [arg1 [arg2 ...]]
```

If you use the `-systf` or `-predefined` command option, the option must appear before the task or function name or the simulator interprets it as an argument to the task or function. See “[call Command Modifiers and Options](#)” on page 506 for details on these options.

The *task\_or\_function\_name* argument is the name of the system task or function with or without the beginning dollar sign. The dollar sign character has a special meaning in Tcl. If the name of the task or function contains any dollar signs, you must enclose the argument in curly braces or precede each dollar sign by a backslash. For example, you can invoke a system task or function called `$mytask` with:

```
ncsim> call mytask  
ncsim> call \$mytask  
ncsim> call {\$mytask}  
ncsim> call {mytask}
```

You can invoke a system task or function called `$my$task` with any of the following:

```
ncsim> call my\$task  
ncsim> call \$my\$task  
ncsim> call {\$my\$task}  
ncsim> call {my\$task}
```

Arguments to the system task or function can be either literals or names.

Literals can be:

■ **Integers**

```
ncsim> call mytask 5  
ncsim> call mytask 5 7
```

■ **Reals**

```
ncsim> call mytask 3.4  
ncsim> call mytask 22.928E+10
```

■ Strings

Strings must be enclosed in double quotes. Enclose strings in curly braces or use the backslash character to escape quotes, spaces, and other characters that have special meaning to Tcl. For example:

```
ncsim> call mytask {"hello world"}  
ncsim> call mytask \"hello\ world\"
```

■ Verilog literals, such as 8'h1f

Names can be full or relative path names of instances or objects. Relative path names are relative to the current debug scope (set by the `scope` command). Object names can include a bit select or part select. For example:

```
ncsim> call mytask top.ul  
ncsim> call mytask top.ul.reg[3:5]
```

Expressions that include operators or function calls are not allowed. For example, the following two commands result in an error:

```
ncsim> call \$mytask a+b  
ncsim> call \$mytask {func a}
```

However, literals can be created using Tcl's `expr` command. For example, if the desired argument is the expression `(a+b)`, use the following:

```
ncsim> call \$mytask [expr #a + #b]
```

The result of the expression `(a+b)` is substituted on the command line and then treated by the `call` command as a literal.

**Note:** The `expr` command cannot evaluate calls to Verilog functions.

If you are calling a user-defined system function, the result of the `call` command is the return value from the system function. Therefore, user-defined system functions can be used to generate literals for other commands. For example:

```
ncsim> call task [call func arg1 ...]  
ncsim> force a = [call func arg1 ...]
```

## call Command Modifiers and Options

The `call` command has two options: `-systf` and `-predefined`.

### **`-systf [task_or_function_name]`**

Look for the specified task or function name only in the table of user-defined PLI system tasks and functions.

This option is available because the `call` command is also used to invoke functions from the VHDL C-interface, and there may be a user-defined C-interface function with the same name as a PLI system task or function. The `-systf` option causes the lookup in the C-interface task list to be skipped.

This option must appear before the task or function name on the command line.

You cannot use this option with the `-predefined` option.

The command `call -systf` with no task or function name argument displays a list of all registered user-defined system tasks and functions.

### **`-predefined [function_name]`**

Look for the specified task or function name only in the table of predefined CFC library functions.

You cannot use the `-predefined` option when calling a user-defined system task or function.

This option must appear before the CFC function name on the command line.

You cannot use this option with the `-systf` option.

The command `call -predefined` with no function name argument displays a list of all predefined C function names.

## call Command Examples

The following Verilog module contains a call to a user-defined system task and to a system function. The task and function can also be invoked from the command line.

```
module test();
initial
begin
    $hello_task();
    $hello_task($hello_func());
end
endmodule
```

The following command invokes the \$hello\_task system task:

```
ncsim> call \$hello_task
```

This task can also be invoked with any of the following:

```
ncsim> call hello_task
ncsim> call {$hello_task}
ncsim> call {hello_task}
```

The \$hello\_func function can be invoked with any of the following commands:

```
ncsim> call \$hello_func
ncsim> call hello_func
ncsim> call {$hello_func}
ncsim> call {hello_func}
```

In the following command, the call command calls the \$hello\_task system task with a call to the system function \$hello\_func as an argument.

```
ncsim> call hello_task [call hello_func]
```

The following command displays a list of all registered user-defined system tasks and functions.

```
ncsim> call -systf
```

## check

The `check` command checks for bus contention and bus float conditions for specified VHDL bus signals (a signal that has multiple drivers). You can use this command only on `std_logic` bus signals. Checks cannot be applied on signals that are declared in a VITAL Level0 scope.

The `check` command performs the bus contention and bus float checks at the following times:

- If any of the drivers of the bus signal change.
- At the end of the simulation.

If the simulator detects a bus contention or bus float, it issues an error message. For a bus contention, the message includes the name of the bus signal on which the contention was detected, the names of the drivers and their current values, the time at which the contention started, and the time at which the time window was exceeded. For a bus float detection, the message includes the bus signal name, the time at which the float started, and the time at which the time window was exceeded.

See “[Checking for Bus Contention and Bus Float Conditions](#)” on page 438 for more information on bus contention and bus float checks.

### check Command Syntax

```
check
  -delay time_limit |
  {-contention | float} signal_specifier [-name check_name] [-delay time_limit]

  -delete check_name [check_name ...]

  -disable check_name [check_name ...]

  -enable check_name [check_name ...]

  -show [check_name ...]
```

## check Command Modifiers and Options

This section describes the modifiers and options that you can use with the `check` command.

### **-contention *signal\_specifier***

Specifies bus contention detection.

The *signal\_specifier* argument can be:

- `-all`
  - Specifies bus contention checking on all bus signals in the current scope.
- A signal name or a list of signal names separated by spaces.
- `-depth all | n scope_name`
  - ❑ `-depth all` specifies bus contention checking on all bus signals in the specified scope and in all scopes in the hierarchy below the specified scope.
  - ❑ `-depth n` specifies bus contention checking on all bus signals in the specified scope and in the specified number of subscopes. For example, `-depth 1` means only the specified scope, `-depth 2` means the specified scope and its subscopes, and so on. The default is 1.

Use the `-name` option to specify a name for the check. By default, checks are numbered sequentially.

Include the `-delay` option to specify the bus contention time limit. If you do not specify a time limit in the `check -contention` command, the time value is the value set with a previous `check -delay` command. If you have not specified a time limit using a `check -delay` command, the default value is 1 fs.

### **-delay *time\_limit***

Specifies the time limit for bus contention or bus float checks.

You can set the time limit for bus contention or bus float checks in two ways:

- Set a default time limit with a `check -delay` command. For example:  

```
ncsim> check -delay 10 ns
```
- Set a time limit for a specific check by including the `-delay` option on a `check -contention` or a `check -float` command. For example:  

```
ncsim> check -float pin1 -delay 10 ns
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

If you do not specify a time limit in the `check -contention` or `check -float` command, the time value is the value set with a previous `check -delay` command. If you have not specified a time limit using a `check -delay` command, the default value is 1 fs.

Changing the default time limit using a `check -delay` command does not affect existing checks.

The bus contention or bus float time limit cannot be smaller than, or more precise than, the unit of simulation.

#### **-delete *check\_name* [*check\_name* ...]**

Deletes the check that has the specified name. If you specify more than one check, separate the names with a space.

You can use wildcard characters (\* or ?) in a `check -delete` command.

#### **-disable *check\_name* [*check\_name* ...]**

Disables the check that has the specified name. If you specify more than one check, separate the names with a space.

You can use wildcard characters (\* or ?) in a `check -disable` command.

To resume checking, use the `-enable` modifier.

#### **-enable *check\_name* [*check\_name* ...]**

Enables a previously disabled check so that checking is resumed. If you specify more than one check, separate the names with a space.

You can use wildcard characters (\* or ?) in a `check -enable` command.

#### **-float *signalSpecifier***

Specifies bus float detection.

The *signalSpecifier* argument can be:

- `-all`

Specifies bus float checking on all bus signals in the current scope.

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

- A signal name or a list of signal names separated by spaces.
- `-depth all | n scope_name`
  - `-depth all` specifies bus float checking on all bus signals in the specified scope and in all scopes in the hierarchy below the specified scope.
  - `-depth n` specifies bus float checking on all bus signals in the specified scope and in the specified number of subscopes. For example, `-depth 1` means only the specified scope, `-depth 2` means the specified scope and its subscopes, and so on. The default is 1.

Use the `_name` option to specify a name for the check. By default, checks are numbered sequentially.

Include the `_delay` option to specify the bus float time limit. If you do not specify a time limit in the `check -float` command, the time value is the value set with a previous `check -delay` command. If you have not specified a time limit using a `check -delay` command, the default value is 1 fs.

The bus contention or bus float time limit cannot be smaller than, or more precise than, the unit of simulation.

#### **`-name check_name`**

Specifies a user-defined name for the check. You can then use the name that you assign to the check with the `-disable`, `-enable`, `-delete`, and `-show` modifiers.

Because the `check -delay` command does not create a check, no check name is created. The following command results in an error:

```
ncsim> check -delay 10 ns -name my_check
```

#### **`-show [check_name ...]`**

Displays information about the check that has the specified name. If you specify more than one check, separate the names with a space.

If you do not include a `check_name` argument, the `check -show` command displays information on all checks.

## check Command Examples

The following VHDL source file is used for the examples in this section:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_textio.all;
use std.textio.all;
entity drvio is
    port ( dr : inout std_logic_vector(1 to 3));
enddrvio;
architecture drv_arch of drvio is
    signal a,b,c : std_logic := 'Z';
begin
    dr(2) <= b;
    dr(2) <= c;
end;

library ieee;
use ieee.std_logic_1164.all;
entity drvo is
    port ( dr : out std_logic_vector(1 to 3));
end drvo;
architecture drv_arch of drvo is
    signal c : std_logic := 'Z';
begin
    dr(2) <= c,'H' after 5 ns ;
    dr(2) <= c, 'L' after 5 ns;
end;

library ieee;
use ieee.std_logic_1164.all;
use ieee.vital_timing.all;
use ieee.vital_primitives.all;
use ieee.std_logic_textio.all;
use std.textio.all;
entity trace is
end trace;
architecture arch of trace is
    component drvio
        port( dr : inout std_logic_vector(1 to 3));
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
end Component;
component drvo
    port( dr : out std_logic_vector(1 to 3));
end Component;
for all: drvio use entity work.drvio(drv_arch);
for all: drvo use entity work.drvo(drv_arch);

signal a, b, c, d, k : std_logic := 'Z';
begin
    a <= d ;
    vitalident(a,b);
    U1 : drvo port map (dr(2) => a);
    U2 : drvio port map (dr(2)=>a);
    a <= 'L' , 'Z' after 2 ns;
    b <= '1' after 4 ns ;
    d <= 'H' after 1 ns,'Z' after 2.5 ns, '0' after 4 ns;
    k <= b;
    k <= c;
process
    begin
        wait for 20 ns;
        assert false severity failure;
    end process;
end;
```

In the following sequence of commands, the `check -delay` command sets a default time limit of 10 fs for bus contention and bus float checks.

The next command creates a bus contention check on bus signal :a. If more than one driver of a is driving a non-Z value for more than 10 fs, bus contention messages will be displayed.

The third command creates a bus float check on bus signal :k. The `-delay` option is included to set a time limit of 20 fs for this check. If no driver is driving k for more than 20 fs, a bus float message will be displayed.

```
% ncvhdl -nocopyright test.vhd
% ncelab -nocopyright worklib.trace:arch
% ncsim -tcl -messages worklib.trace:arch
ncsim: v3.20.(p1): (c) Copyright 1995 - 2000 Cadence Design Systems, Inc.
Loading snapshot worklib.trace:arch ..... Done
ncsim> check -delay 10 fs
ncsim> check -contention :a
Created check 1
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
ncsim> check -float :k -delay 20 fs
Created check 2
ncsim> run
2 NS: Contention detected on signal :a(test.vhd,46) at 1000010 FS from 1 NS between
the following drivers -
value --> 'L': a <= 'L' , 'Z' after 2 ns [File: test.vhd, Line: 52]
value --> 'H': a <= d [File: test.vhd, Line: 48]
4 NS: Float detected on signal :k(test.vhd,46) at 20 FS from 0 FS
5 NS: Contention detected on signal :a(test.vhd,46) at 4000010 FS from 4 NS between
the following drivers -
value --> '1': vitalident(a,b) [File: test.vhd, Line: 49]
value --> '0': a <= d [File: test.vhd, Line: 48]
ASSERT/FAILURE (time 20 NS) from process :$PROCESS_007 (architecture
WORKLIB.trace:arch)
Assertion violation.
Assertion at 20 NS + 0
./test.vhd:60      assert false severity failure;
```

The following `reset` command reloads the snapshot. The `check -show` command shows that the checks are still enabled.

```
ncsim> reset
Loaded snapshot worklib.trace:arch
ncsim> check -show
1     Enabled          Time Limit      10 FS:a      -contention
2     Enabled          Time Limit      20 FS:k      -float
```

The following command deletes all checks.

```
ncsim> check -delete *
Deleted Check *
ncsim> check -show
No checks set
```

The following command specifies bus contention detection on bus signals `:a` and `:u1:dr(2)`. The `-name` option is included to specify a user-defined name for the check.

```
ncsim> check -contention :a :u1:dr(2) -name contention_check
Created check contention_check
```

The following command specifies bus float detection on all bus signals in the current scope and its subscopes. The `-name` option is included to specify a user-defined name for the check.

```
ncsim> check -float -depth 2 : -name float_check
ncsim: *W,CHSBPR: check can not be applied on :U1:DR(1).
ncsim: *W,CHSBPR: check can not be applied on :U1:DR(3).
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
ncsim: *W,CHSBPR: check can not be applied on :U2:DR(1).
ncsim: *W,CHSBPR: check can not be applied on :U2:DR(3).
Created check float_check
```

The following command displays information on the check named contention\_check.

```
ncsim> check -show contention_check
contention_check      Enabled      10 FS      :a      -contention
                      :U1:dr(2)
```

The following command disables the check named contention\_check.

```
ncsim> check -disable contention_check
Disabled Check contention_check
ncsim> check -show
float_check      Enabled      10 FS      :      -float -depth 2
:
:
contention_check      Disabled      10 FS      :a      -contention
                      :U1:dr(2)

ncsim> run
4 NS: Float detected on signal :k(test.vhd,46) at 10 FS from 0 FS
4 NS: Float detected on signal :a(test.vhd,46) at 2500010 FS from 2500 PS
5 NS: Float detected on signal :U1:dr(2)(test.vhd,19) at 10 FS from 0 FS
ASSERT/FAILURE (time 20 NS) from process :$PROCESS_007 (architecture
WORKLIB.trace:arch)
Assertion violation.
Assertion at 20 NS + 0
./test.vhd:60      assert false severity failure;
```

Checks are performed if any of the drivers of the bus signal change and at the end of the simulation.

```
ncsim> exit
20 NS: Float detected on signal :U2:dr(2)(test.vhd,6) at 10 FS from 0 FS
%
```

## coverage

The `coverage` command controls the dumping of code coverage data. You must use the `ncelab -coverage` option to enable code coverage.

**Note:** Code coverage is not currently available on Windows platforms.

Three types of coverage are supported:

- Statement coverage, which tells you whether a particular statement was executed or was not executed during a given simulation run. The code coverage analyzer reports the statement coverage statistic in boolean format, which reports 1 if the statement has executed and 0 if the statement has not executed.
- State machine (fsm) coverage, which tells you which states the state machine variables have been in (state visitation), and which state-to-state transitions have taken place (state transitions).

To enable state machine coverage, you must use the `ncelab -access +r` option (`ncverilog +ncaccess+r`) to provide read access to simulation objects.

- Expression coverage, which identifies which terms of an expression in your code contribute to true and false values for the expression's terms during a simulation run.

To enable expression coverage, you must compile the Verilog modules (and VHDL design units) with the `-linedebug` option.

**Note:** Using the expression recording features can severely impact your simulation performance.

Code coverage generates results files in a code coverage database directory. After using the `coverage` command to generate the coverage database, you can read the results into the coverage analyzer, NC-Cov, which lets you examine the coverage results. You can analyze your coverage results interactively, generate batch reports, and merge coverage results databases.

Code coverage analysis is supported for both Verilog and VHDL. Code coverage for mixed-language designs is supported in the NC-Sim mixed language simulator.

See the *Code Coverage User Guide* for details on code coverage and for information on the syntax and options for the `coverage` command.

## database

The `database` command lets you control an SHM, Value Change Dump (VCD), or Extended Value Change Dump (EVCD) database. You can:

- Open a database by using the optional `-open` modifier. You can open the following kinds of databases:
  - SHM (for Verilog, VHDL, or mixed-language)
  - VCD (for Verilog, VHDL, or mixed-language)
  - EVCD (for VHDL only).
- For Verilog you must use the `$dumpports` system task. See “[Generating an Extended Value Change Dump \(EVCD\) File](#)” on page 463.
- Set a database as the default database (`database -setdefault`)
- Display information about databases (`database -show`).
- Disable databases (`database -disable`).
- Enable databases (`database -enable`).
- Close a database (`database -close`).

See “[Managing Databases](#)” on page 418 for more information.

After opening an SHM, VCD, or EVCD database, you can then probe the items that you want to dump to the database by using the `probe` command. See “[probe](#)” on page 571 for information on probing objects to a database with the `probe` command.

For Verilog, in addition to creating an SHM or a VCD database with the `database` and `probe` commands, you can:

- Create an SHM database by using the `$shm_open` and `$shm_probe` system tasks in your Verilog source code. See “[Creating an SHM Database and Probing Signals](#)” on page 440 for details.

For backward compatibility with Verilog code that contains calls to the Signalscan tasks for recording data (`$recordvars`, `$recordfile`, `$recordsetup`, and so on), Cadence has implemented these tasks as system tasks native to the simulator. See “[Using \\$recordvars and Related Tasks](#)” on page 446 for details.
- Create a VCD database by using value change dump system tasks (`$dumpfile`, `$dumpvars`, `$dumpall`, and so on) in your Verilog source code. See “[Generating a Value Change Dump \(VCD\) File](#)” on page 458 for more information.

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

To create an EVCD database, you must use the `$dumpports` system task. See “[Generating an Extended Value Change Dump \(EVCD\) File](#)” on page 463.

For VHDL, in addition to creating a VCD database with the `database` and `probe` commands, you can also open a VCD database and probe objects to the database by using the `call` command to call predefined CFC routines, which are part of the NC-VHDL simulator C interface. This feature has been retained for backwards compatibility. The recommended method of generating a VCD file is to open a database with the `database -open -vcd` command and to probe objects to the database with the `probe -vcd` command. See the appendix called “[Generating a VCD File Using CFC Routines](#)” in the *NC-VHDL Simulator Help* for more information. See “[call](#)” on page 504 for details on the `call` command.

For information on generating an EVCD database for VHDL, see the section called “[Generating an Extended Value Change Dump \(EVCD\) File](#)” in the chapter called “[Debugging Your Design](#)” in the *NC-VHDL Simulator Help*.

If you are doing transaction-based verification, which is available with the Cadence® Verification Cockpit, you can have only one database open when dumping transactions. See the [SimVision Waveform Viewer User Guide](#) for information on transaction recording and viewing.

## database Command Syntax

```
database
  [-open] dbase_name
    [-compress]
    [-default]
    [-evcd] [vector] [-timescale timescale_value]
    [-event]
    [-into filename]
    [-maxsize max_byte_size]
    [-shm]
    [-vcd] [-timescale timescale_value]

  -close {dbase_name | pattern} ...
  -disable {dbase_name | pattern} ...
  -enable {dbase_name | pattern} ...
  -setdefault dbase_name [dbase_name ...]
  -show [{dbase_name | pattern} ...]
```

The argument to `-open` is a database name.

The argument to `-close`, `-disable`, `-enable`, or `-show` can be:

- A database name
- A list of database names
- A pattern
  - The asterisk ( `*` ) matches any number of characters.
  - The question mark ( `?` ) matches any one character.
  - `[characters]` matches any one of the characters.
- Any combination of literal database names and patterns

## database Command Modifiers and Options

### Opening a Database

#### **[-open] *dbase\_name***

Creates a new database with the name specified by the *dbase\_name* argument. The *-open* modifier is optional.

By default, the `database` command opens an SHM database. Use the `-vcd` option to open a VCD database or the `-evcd` option to open an EVCD database.

#### **-compress**

Compresses an SHM database to reduce its size. This option has no effect on VCD or EVCD databases.

Data is stored in the SHM database by signal, and the data for each signal is compressed independently. This allows data for arbitrary signals to be loaded and viewed without uncompressing an entire file. Only the data that is being loaded is uncompressed, and this is done automatically as it is read.

Signal transition data is always compressed. The default level of compression requires no additional memory, and the compression time is minimal. The time spent in compression is, in almost all cases, made up by decreased I/O time, since less data is written to disk.

The `-compress` option enables maximum compression. This requires additional memory and takes more compression time, but results in a smaller database.

The amount of time spent in compression, and the resulting size of the database, is highly dependent on the characteristics of the data. For example, with the `-compress` option, data that is essentially random may take extra time to write as the program unsuccessfully searches for patterns in the data, and the database may not use significantly less disk space. Without the `-compress` option, random data does not require any additional time other than the time to write that data to disk.

The `-compress` option may also not decrease the database size significantly if all of the data consists of very simple patterns that are found using the fast compression that is enabled by default. Most of the time, however, the data consists of a mixture of simple patterns, more complex patterns, and uncompressible data, and `-compress` will produce a smaller database at the cost of some additional time and memory.

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

There is no penalty when reading the database if `-compress` was used to write the data. In fact, reading will be faster if the compression was effective because there is less data that must be read from disk.

Because the size of the tradeoff is highly dependent on the data, you might want to try both levels of compression. In general, however, if you want the highest writing speed or lowest memory use, use the default level of compression. If you want minimum database size, use `-compress`.

#### **-default**

Specifies that this is the default database for all signal tracing of the same kind (SHM, VCD, or EVCD).

#### **-evcd**

**Note:** You cannot generate an EVCD file for Verilog by using Tcl commands. For Verilog, you must use the `$dumpports` system task to generate an EVCD file.

Specifies that this is an EVCD database.

Include the `vector` parameter after the `-evcd` option if you want to dump whole vectors. By default, elements of vector ports are dumped individually.

The `-evcd` option can be followed by the `vector` parameter or by another command-line option. You cannot specify the database name immediately after the `-evcd` option. The following command generates an error:

```
ncsim> database -evcd mydatabase
```

#### **-event**

Dumps all value changes to the database.

By default, when probing to an SHM database, the simulator discards multiple value changes for an object during one simulation time and dumps only the final value at the end of that simulation time. Use `-event` if you want to dump all value changes to the SHM database. You can then use SimVision Waveform Viewer to expand a single moment of simulation time to show the sequence of value changes that occurred at that time.

This option is not turned on for a database when a `probe` command causes the automatic creation of a default database.

This option has no effect on VCD or EVCD databases. The simulator always dumps all value changes to a VCD or EVCD database.

If you are doing transaction-based verification, you must use the `-event` option to enable transaction recording. See the [\*SimVision Waveform Viewer User Guide\*](#) for information on transaction recording and viewing.

***-into filename***

Specifies the physical filename for the database. By default, the filename for an SHM database is `dbase_name.shm`, the filename for a VCD database is `dbase_name.vcd`, and the filename for an EVCD database is `dbase_name.evcd`. Use the `-into` option to override these defaults.

***-maxsize max\_byte\_size***

Sets a limit on the size of the database. By default, there is no size limit.

The `max_byte_size` argument must be a positive integer.

For VCD and EVCD databases, this option specifies a limit on the number of bytes that the simulator can dump to the VCD or EVCD file. This option has the same effect as the Verilog `$dumplimit` system task for VCD. If the size of the VCD or EVCD file reaches the specified limit, the simulator inserts a comment (`dump limit reached`) into the file, and dumping stops.

In most cases, the VCD or EVCD output file size will be no more than a few hundred bytes over the specified limit. However, because the header and initial values are always dumped regardless of the limit and because the limit is not checked until after the header and initial values are written, the size of the database could far exceed the limit specified using the `-maxsize` option if you have a large design in which the number of objects that you are probing is very high.

SHM (SST2) databases exist as chunks that vary in size from about 2Mb to about 4Mb. Thus, the minimum possible size of the database is somewhere between 2Mb and 4Mb. If you set the maximum size to less than this approximate range of sizes, the resulting database size can still be up to about 4Mb.

When the maximum size that you specified with the `-maxsize` option is about to be exceeded, the simulator maintains the size limit by discarding an entire chunk (2-4 MB) of the earliest recorded values. This means that if the maximum size that you specify is greater than 4Mb, the database size will be below the limit, but it can be up to 4Mb below the limit.

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

When the size limit is exceeded, the waveform window displays an "unknown" value for each object from the beginning of the simulation to the time of the first undiscarded value.

If a `probe` command automatically creates a default database, the `-maxsize` option has no effect.

#### **-shm**

Specifies that this is an SHM database. This is the default.

#### **-timescale *timescale\_value***

Sets the `$timescale` value in the VCD or EVCD file to the specified timescale. This option lets you output a different timescale in the VCD or EVCD file than the timescale being used during simulation. In the output file, the times that are shown for the signal changes reflect the simulation times at the precision that you specify with the `-timescale` option.

The *timescale\_value* argument can be:

- fs, 10fs, 100fs
- ps, 10ps, 100ps
- ns, 10ns, 100ns
- us, 10us, 100us
- ms, 10ms, 100ms

You can use the `-timescale` option only with the `-vcd` or `-evcd` option.

Examples:

```
database -open -vcd test_mp -timescale 10ps
database -open -evcd test_mp -timescale ns
```

#### **-vcd**

Specifies that this is a VCD database.

## Setting a Database As the Default

### **-setdefault *dbase\_name* [*dbase\_name* ...]**

Makes the specified database(s) the default database for its kind. For example, the following command makes the database `waves.shm` the default SHM database.

```
ncsim> database -setdefault waves.shm
```

This modifier is useful if you want to specify that a previously opened database is now to be used as the default database for probes and other operations. For example, suppose that you open a database with the following `$shm_open` task in your Verilog HDL:

```
$shm_open( "waves.shm" );
```

This opens a database called `_waves.shm` (the underscore character indicates that the database was opened from within the HDL). If you then want to add additional probes to this database, you can:

- Use the probe `-database _waves.shm` command. The `-database` option specifies that you want the data saved in `_waves.shm`. In SimVision, use the *Set—Probe* command and specify the database on the Set Probe form.
- Use the `-setdefault` modifier to make `_waves.shm` the default SHM database, and then probe the signals. If you do not make `_waves.shm` the default database, the simulator will open a default SHM database called `ncsim.shm` for you, and the signals will be probed to that database.

## Displaying Information about Databases

### **-show [ {*dbase\_name* | *pattern*} ... ]**

Displays information about the database(s) specified by the argument. If you do not specify an argument, the simulator displays information about all open databases.

## Disabling a Database

### **-disable { *dbase\_name* | *pattern* } ...**

Disables the tracing of data into the database(s) specified by the argument.

## Enabling a Database

**-enable { *dbase\_name* | *pattern* } ...**

Resumes the tracing of data into the database(s) specified by the argument.

## Closing a Database

**-close { *dbase\_name* | *pattern* } ...**

Closes the database(s) specified by the argument.

## database Command Examples

The following command opens an SHM database named `waves` and places the data into the file `waves.shm`. The `-open` modifier is optional. The `-shm` option is the default.

```
ncsim> database -open waves -shm  
Created SHM database waves
```

The following command opens an SHM database named `waves` and places the data into the file `mywaves.shm`.

```
ncsim> database waves -into mywaves.shm  
Created SHM database waves
```

The following command opens an SHM database named `waves` and places the data into the file `waves.shm`. The `-default` option specifies that `waves` is the default database.

```
ncsim> database waves -default  
Created default SHM database waves
```

The following command includes the `-compress` option, which compresses the SHM database.

```
ncsim> database -open waves -compress -default  
Created default SHM database waves
```

The following command opens a VCD database named `vcddb` and places the data into the file `vcddb.vcd`.

```
ncsim> database -open vcddb -vcd  
Created VCD database vcddb
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following command opens a VCD database named vcddb. The `-timescale` option sets the `$timescale` value in the VCD file to 1 ns. Value changes in the VCD file are scaled to 1 ns.

```
ncsim> database -open -vcd vcddb -timescale ns
Created VCD database vcddb
```

The following command opens an EVCD database named evcddb and places the data into the file evcddb.evcdb.

```
ncsim> database evcddb -evcd
Created EVCD database evcddb
```

The following command displays information about the status of all databases.

```
ncsim> database -show
```

The following command displays information about the status of all databases that have names that start with db.

```
ncsim> database -show db*
```

The following command displays information about the status of all databases that have names that start with db and end with 1.

```
ncsim> database -show db*1
```

The following command displays information about the status of all databases that have names that start with v or w.

```
ncsim> database -show v* w*
```

The following command is identical to the previous command. It displays information about the status of all databases that have names that start with v or w. The curly braces suppress command substitution with square brackets.

```
ncsim> database -show {[vw]}*
```

The following command displays information about the status of all databases that have names that have three characters, starting with db.

```
ncsim> database -show db?
```

The following command displays information about the status of the database called waves and of all databases with names that start with db.

```
ncsim> database -show waves db*
```

The following command disables the databases named db1 and db2.

```
ncsim> database -disable db1 db2
```

## **NC-Verilog Simulator Help**

### Using the Tcl Command-Line Interface

---

The following command enables the database named db1.

```
ncsim> database -enable db1
```

The following command closes all databases that have names that start with db.

```
ncsim> database -close db*
```

## deposit

The `deposit` command lets you set the value of an object. Behaviors that are sensitive to value changes on the object run when the simulation resumes, just as if the value change was caused by the Verilog or VHDL code.

The `deposit` command without a delay is similar to a force in that the specified value takes effect and propagates immediately. However, it differs from a force in that future transactions on the signal are not blocked.

You can specify that the deposit is to take effect at a time in the future (`-after -absolute`) or after some amount of time has passed (`-after -relative`). In VHDL, a deposit with a delay is different from Verilog in that it creates a transaction on a driver, much the same as a VHDL signal assignment statement. Use the `-inertial` or `-transport` option to deposit the value after an inertial delay or after a transport delay, respectively.

For VHDL, you can deposit to ports, signals, and variables if no delay is specified. If a delay is specified, you cannot deposit to variables or to signals with multiple sources.

For Verilog, you can deposit to ports, signals (wires and registers), and variables.

If the object is a memory or a range of memory elements, the specified value is deposited into each element of the memory or into each element in the specified range.

If the object is currently forced, the specified value appears on the object after the force is released, unless the release value is overwritten by another assignment in the meantime.

If the object is a register that is currently forced or assigned, the `deposit` command has no effect.

The value assigned to the object must be a literal. The literal can be generated with Tcl value substitution or command substitution. (See “[Value Substitution](#)” on page 979 and “[Command Substitution](#)” on page 979 for details on Tcl substitution.)

For VHDL, the value specified with the `deposit` command must match the type and subtype constraints of the VHDL object. Integers, reals, physical types, enumeration types, and strings (including `std_logic_vector` and `bit_vector`) are supported. Records and non-character array values are not supported, but objects of these types can be assigned to by issuing commands for each subelement individually.

The object to which the value is to be deposited must have read/write access. An error is generated if the object does not have this access. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details on specifying access to simulation objects.

## **deposit Command Syntax**

```
deposit object_name [=] value
      [-after time_spec {-relative | -absolute} ]
      [-inertial]
      [-transport]
```

## **deposit Command Modifiers and Options**

### **-after *time\_spec***

Causes the assignment to occur at a time in the future, rather than immediately. The time specified in the *time\_spec* argument can be relative (the default) or absolute.

If you do not specify a time, the assignment happens immediately, before simulation resumes. If the specified time is the current simulation time, the assignment occurs after simulation resumes, but before time advances.

### **-absolute**

Causes the assignment to occur at the simulation time specified in the *time\_spec* argument.

### **-relative**

Causes the assignment to occur after the amount of time specified in the *time\_spec* argument has passed. This is the default.

### **-inertial**

Deposits the value after an inertial delay.

### **-transport**

Deposits the value after a transport delay.

## deposit Command Examples

### Verilog examples:

The following command assigns the value 8'h1F to r[0:7]. No time for this assignment is specified, so the assignment occurs immediately. The equal sign is optional.

```
ncsim> deposit r[0:7] = 8'h1F
```

The following command assigns 25 to r[8:15] after simulation resumes and 1 time unit has elapsed.

```
ncsim> deposit r[8:15] = 25 -after 1
```

The following command assigns 25 to r[8:15] at simulation time 1 ns.

```
ncsim> deposit r[8:15] = 25 -after 1 ns -absolute
```

The following command sets the value of x to the current value of w. The assignment occurs at simulation time 10 ns.

```
ncsim> deposit x = #w -after 10 ns -absolute
```

The following command uses both command and value substitution. The object y is set to the value returned by the Tcl `expr` command, which evaluates the expression `#r[0] & ~#r[1]` using the current value of r.

```
ncsim> deposit y = [expr #r[0] & ~#r[1]]
```

The following command shows the error message that is displayed if you run in regression mode and then try to deposit a value to an object that does not have read/write access.

```
ncsim> deposit clrb 1  
ncsim: *E,RWACRQ: Object does not have read/write access: harddrive.h1.clrb.
```

### VHDL examples:

The following command deposits the value 1 to object :t\_nickel\_out (std\_logic). The equal sign is optional.

```
ncsim> deposit :t_nickel_out = '1'
```

The following command deposits the value 1 to object :top:DISPENSE\_tempsig (std\_logic).

```
ncsim> deposit :top:DISPENSE_tempsig '1'
```

The following command deposits the value 0 to object :t\_dimes (std\_logic\_vector) after 10 ns has elapsed.

```
ncsim> deposit -after 10 ns -relative :t_DIMES {"00000000"}
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following command deposits the value TRUE to object `stoppit` (boolean).

```
ncsim> deposit stoppit true
```

The following command deposits the value 10 to object `:count` (integer).

```
ncsim> deposit :count 10
```

You can deposit values to arrays and records only by depositing to each individual element separately. For example, suppose that you have defined the following record:

```
TYPE record_type is RECORD
  var1: INTEGER;
  var2: STD_LOGIC;
  var3: BOOLEAN
END RECORD
record1: record_type;
```

The following commands can be used to deposit values to the elements of this record.

```
ncsim> deposit record1.var1 0
ncsim> deposit record1.var2 '0'
ncsim> deposit record1.var3 true
```

In the following example, an array is defined. This is followed by two `deposit` commands that deposit values to the elements of the array.

```
TYPE iclk_time_type is ARRAY(1 DOWNTO 0) of time;
iclk_period: iclk_time_type;

ncsim> deposit iclk_period[1] 10 ns
ncsim> deposit iclk_period[0] 5 ns
```

## describe

The `describe` command displays information about the specified simulation object, including its declaration.

- For objects without read access, the output of the `describe` command does not include the object's value.
- For objects that have read access but no write access, the string (`-W`) is included in the output.
- For objects with neither read nor write access, the string (`-RW`) is included in the output.

See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details on specifying access to simulation objects.

### describe Command Syntax

```
describe simulation_object ...
```

You can use wildcard characters in the argument to a `describe` command.

- The asterisk (`*`) matches any number of characters.
- The question mark (`?`) matches any one character.

You cannot use wildcard characters inside escaped names.

See “[Using Wildcards Characters in Tcl Commands](#)” on page 495 for more information on using wildcards.

### describe Command Modifiers and Options

None.

## describe Command Examples

### Verilog examples:

The following command displays information about the Verilog object `data`.

```
ncsim> describe data
data.....register [3:0] = 4'h0
```

The following command displays information about two Verilog objects: `data` and `q`.

```
ncsim> describe data q
data.....register [3:0] = 4'h0
q.....wire [3:0]
q[3] (wire/tri) = StX
q[2] (wire/tri) = StX
q[1] (wire/tri) = StX
q[0] (wire/tri) = StX
```

The following command displays information about all objects in the current scope that have names that start with the letter `c`.

```
ncsim> describe c*
count.....wire [3:0]
count[3] (wire/tri) = St1
count[2] (wire/tri) = St0
count[1] (wire/tri) = St1
count[0] (wire/tri) = St0
clock.....wire (wire/tri) = St0
```

The following command displays information about all objects in the current scope that have names that are six characters long and that start with `rst` and end with `p5`.

```
ncsim> describe rst?p5
rst7p5.....register = 1'h0
rst6p5.....register = 1'h0
rst5p5.....register = 1'h0
```

The following command displays information about the object `alu_16`.

```
ncsim> describe alu_16
alu_16.....top-level module
```

The following command displays information about the object `u1`.

```
ncsim> describe u1
u1.....instance of module arith
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following command shows the output of the `describe` command for an object that does not have read or write access. The output of the command includes the string ( -RW ) instead of the object's value.

```
ncsim> describe d
d.....input [3:0]
  d[3]  (-RW)
  d[2]  (-RW)
  d[1]  (-RW)
  d[0]  (-RW)
```

#### VHDL examples:

```
ncsim> describe t_NICKEL_IN
t_NICKEL_IN...signal : std_logic = '0'

ncsim> describe t_NICKEL_IN t_CANS
t_NICKEL_IN...signal : std_logic = '0'
t_CANS.....signal : std_logic_vector(7 downto 0) = "11111111"

ncsim> describe :top
top.....component instantiation

ncsim> describe :t_DI*
:t_dime_outsignal : std_logic = '0'
:t_dispenseignal : std_logic = '0'
:t_dimes...signal : std_logic_vector(7 downto 0) = "11111111"
:t_dime_in.signal : std_logic = '0'
```

In the following example, the `stack -show` command displays the current call stack. The process (`process1`) is displayed as nest-level 0, the base of the stack. The subprogram `function1` is `:process1[1]`, and the subprogram `function2` is `:process1[2]`.

- The first `describe` command describes the object `tmp5_local`, which is in the current debug scope.
- The second `describe` command describes the object `tmp4` in `:process1[1]`.
- A `scope` command is then executed to set the scope to `:process1[1]`. Because the current debug scope is now `function1`, you can refer to object `tmp4` by simply using its name.
- The last `describe` command describes `var4` in `:process1`.

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
ncsim> stop -subprogram function1
Created stop 1
ncsim> run
0 FS + 0 (stop 1: Subprogram :function1)
./test.vhd:36 tmp4_local := function2 (tmp4);
ncsim> run -step
./test.vhd:29 tmp5_local := tmp5 + 1;
ncsim> stack -show          ;# Display the current call stack
2: Scope: :process1[2] Subprogram:@work.e(a):function2
   File: /usr1/belanger/inca/vhdl/subprogram_debug/scope_test/test.vhd
   Line: 29

1: Scope: :process1[1] Subprogram:@work.e(a):function1
   File: /usr1/belanger/inca/vhdl/subprogram_debug/scope_test/test.vhd
   Line: 36

0: Scope: :process1
   File: /usr1/belanger/inca/vhdl/subprogram_debug/scope_test/test.vhd
   Line: 52
ncsim> scope -show          ;# Display the current debug scope
Directory of scopes at current scope level:

Current scope is (:process1[2])
Highest level modules:
  Top level VHDL design unit:
    entity (e:a)
  VHDL Package:
    STANDARD
    ATTRIBUTES
    std_logic_1164
    TEXTIO
ncsim> describe tmp5_local    ;# Describe tmp5_local in the current debug scope
tmp5_local...variable : INTEGER
ncsim>
ncsim> describe :process1[1]:tmp4      ;# Describe tmp4 in :process1[1]
                                         ;# i.e., function1
:process1[1]:tmp4...constant parameter : INTEGER
ncsim>
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
ncsim> scope -set :process1[1]      ;# Set scope to function1 (:process1[1])
ncsim> describe tmp4                ;# Describe tmp4 in function1
tmp4.....constant parameter : INTEGER
ncsim>
ncsim> describe :process1:var4
:process1:var4....variable : INTEGER = 0
```

## drivers

The `drivers` command displays a list of all contributors to the value of the specified object(s). You must specify at least one object.

You can use the `scope -drivers [ scope_name ]` command to display the drivers of each object that is declared within a specified scope. See “[scope](#)” on page 608 for details on the `scope` command.

The `drivers` command cannot find the drivers of a wire or register unless the object has read and connectivity access. However, even if you have specified access to an object, its drivers may have been collapsed, combined, or optimized away. In this case, the output of the command may indicate that the object has no drivers. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details on specifying access to simulation objects.

See “[drivers Command Report Format](#)” on page 538 for details on the output format of the `drivers` command. See “[drivers Command Examples](#)” on page 542 for examples.

### drivers Command Syntax

```
drivers object_name ...
  [-effective]
  [-future]
  [-novalue]
  [-verbose]
```

### drivers Command Modifiers and Options

#### **-effective**

Displays contributions to the effective value of the signal. By default, the `drivers` command displays contributions to the driving value.

Only VHDL inout and linkage ports can have different driving and effective values.

#### **-future**

Displays the transactions that are scheduled on each driver.

**-novalue**

Suppresses the display of the current value of each driver.

**-verbose**

**Note:** This option affects VHDL signals only.

Displays all of the processes (signal assignment statements), resolution functions, and type conversion functions that contribute to the value of the specified signal.

If you don't include the `-verbose` option, resolution and type conversion function information is omitted from the output.

## drivers Command Report Format

### Verilog Signals

The drivers report for Verilog signals is as follows:

```
value <- (scope) verilog_source_line_of_the_driver
```

For example:

```
af.....wire (wire/tri) = St1  
St1 <- (board.counter) assign altFifteen = &value
```

Instead of the `verilog_source_line_of_the_driver`, the following is output when the actual driver is from a VHDL model:

```
port 'port_name' in module_name [File:  
path_to_fileContaining_module], driven by a VHDL model.
```

This report indicates that the signal is ultimately driven by a port (connected to `port_name` of the specified module) on a module whose body is an imported VHDL model. The `module_name` corresponds to the module name of the shell being used to import the VHDL model.

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

## VHDL Signals

The drivers report for VHDL signals is as follows:

```
description_of_signal = value
value_contributed_by_driver <- (scope_name) source_description
```

The *source\_description* for the various kinds of drivers are shown below:

### A Process

Nothing is generated for the *source\_description*. This implies that a sequential signal assignment statement within a process is the driver. The *scope\_name* is the scope name of the process.

### Concurrent Signal Assignment/Concurrent Procedure call

The *source\_description* is the VHDL source text of the concurrent signal assignment statement or concurrent procedure call that results in a driving value. This concurrent statement is within the scope *scope\_name*.

### No drivers

If the signal has no drivers, the text No drivers appears verbatim.

### A Verilog driver

If the driver is from a Verilog model, the report has the following form:

```
port 'port_name' in entity(arch) [File:
    path_to_file_containing_entity], driven by a Verilog model.
```

This report indicates that the signal is ultimately driven by a port (connected to *port\_name* of the specified entity-architecture pair) on an entity whose body is an imported Verilog model.

### Driver from a C model

If the driver is from an imported C model, the report has the following form:

```
port 'port_name' in entity(arch) [File:
    path_to_file_containing_entity], driven by a C model.
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

#### **Driver from a LMC model**

If the driver is from an imported LMC model, the report has the following form:

```
port 'port_name' in entity(arch) [File:  
    path_to_file_containing_entity], driven by a LMC model.
```

#### **Driver from an OMI model**

If the driver is from an imported OMI model, the report has the following form:

```
port 'port_name' in entity(arch) [File:  
    path_to_shell_file], driven by a OMI model.
```

#### **Resolution / Type Conversion Function in Non-Verbose mode**

If you don't use the `-verbose` option, the text [ verbose report available .... ] may appear. This indicates that the signal gets its value from a resolution function or a type conversion function. Use `-verbose` to display more information on the derivation of the signal's value.

On the next line of output (indented), a nonverbose driver report is displayed for each signal whose driver contributes to the value of the signal in question.

#### **Resolution Function**

The following text is generated only when the `-verbose` option is used:

```
[resolution function function_name()]
```

This means that the signal is resolved with the named resolution function. A verbose drivers report is displayed (indented) for all inputs to the resolution function.

#### **Type conversion on Formal of Port Association**

The following text is generated only when the `-verbose` option is used:

```
[type conversion function function_name(formal)]
```

This means that the signal's driving value comes from a type conversion function on a formal in a port association. A verbose drivers report is displayed (indented) for the formal port that is the input to the function.

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

#### **Type Conversion on Actual of Port Association**

The following text is generated only when the `-verbose` option is used:

```
[type conversion function function_name(actual)]
```

This means that the signal's effective value comes from a type conversion function on an actual in a port association. A verbose drivers report is displayed (indented) for the actual that is the input to the function.

#### **Implicit Guard Signal**

The following text is displayed in response to a query on a signal whose value is computed from a GUARD expression:

```
[implicit guard signal]
```

#### **Signal Attribute**

The following is displayed in response to a query on an IN port that ultimately is associated with a signal valued attribute:

```
[attribute of signal full_path_of_the_signal]
```

The *full\_path\_of\_the\_signal* corresponds to the complete hierarchical path name of the signal whose attribute is the driver.

#### **Constant Expression on a Port Association**

The following is displayed when the value of the signal in question is from a constant expression in a port map association:

```
[constant expression associated with port port_name]
```

#### **Composite Signals**

For a composite signal, a separate report is displayed for each group of subelements that can be uniquely named and that have the same set of drivers.

## drivers Command Examples

This section includes examples of using the **drivers** command with Verilog and with VHDL signals.

### Example Output for Verilog Signals

The following command lists the drivers of a signal called f.

```
ncsim> drivers f
f.....wire (wire/tri) = StX
    StX <- (board.counter) assign fifteen = value[0] & value[1] & value[2] &
          value[3]
```

The following command lists the drivers of two signals called f and af.

```
ncsim> drivers f af
f.....wire (wire/tri) = StX
    StX <- (board.counter) assign fifteen = value[0] & value[1] &
          value[2] & value[3]
af.....wire (wire/tri) = StX
    StX <- (board.counter) assign altFifteen = &value
```

The following command lists the drivers of a signal called top.under\_test.sum.

```
ncsim> drivers top.under_test.sum
top.under_test.sum...output [1:0] (wire/tri) = 2'h0 (-W)
    2'h0 <- (top.under_test) assign {c_out, sum} = a + b + c_in
```

The following command lists the drivers of a signal called board.count.

```
ncsim> drivers board.count
board.count.....wire [3:0]
    count[3] (wire/tri) = St1
        St1 <- (board.counter.d) output port 1, bit 0 (.counter.v:10)
    count[2] (wire/tri) = St0
        St0 <- (board.counter.c) output port 1, bit 0 (.counter.v:9)
    count[1] (wire/tri) = St1
        St1 <- (board.counter.b) output port 1, bit 0 (.counter.v:8)
    count[0] (wire/tri) = St0
        St0 <- (board.counter.a) output port 1, bit 0 (.counter.v:7)
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following command shows the error message that the simulator displays if you run in the default “regression” mode (no read, write, or connectivity access to simulation objects) and then use the **drivers** command to find the drivers of an object that does not have read and connectivity access.

```
ncsim> drivers count
ncsim: *E,OBJACC: Object must have read and connectivity access: board.count.
```

The following examples illustrates the output of the **drivers** command when the actual driver is from a VHDL model:

```
ncsim> drivers :u1.a
u1.a.....input (wire/tri) = St1
    St1 <- (:u1) driven by a VHDL model

ncsim> drivers :u1.v.d
u1.v.d....input (wire/tri) = St1
    St1 <- (:u1) port 'a' in module 'and2' [File: ./verilog.v],
        driven by a VHDL model
ncsim>
```

This report indicates that the signal `:u1.v.d` is ultimately driven by a port (connected to port `a` of the module `and2`) on a module whose body is an imported VHDL model.

### Example Output for VHDL Signals

The following examples use the VHDL model shown below. A `run` command has been issued after invoking the simulator.

```
library ieee;
use ieee.std_logic_1164.all;
entity e is
end e;
architecture a of e is
    signal s: std_logic;
    function bit_to_std (x: bit) return std_logic is
    begin
        return '0';
    end bit_to_std;
    function std_to_bit (x: std_logic) return bit is
    begin
        return '1';
    end std_to_bit;
begin
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
s <= '0' after 1 ns;
GATE: block
    port (q: inout bit);
    port map (bit_to_std (q) => std_to_bit (s));
begin
    p: process (q)
    begin
        q <= not q;
    end process;
end block;
end;
```

The following command shows the drivers of signal s. The string [verbose report available .....] indicates that type conversion functions or resolution functions are part of the hierarchy of drivers. Use the -verbose option to display this additional information.

```
ncsim> drivers s
s.....signal : std_logic = '0'
[verbose report available.....]
'0' <- (:GATE:p) [File: test.vhd]
'0' <- (:) s <= '0' after 1 ns [File: test.vhd, Line: 20]
```

The following command includes the -novalue option, which suppresses the display of the current value of each driver.

```
ncsim> drivers s -novalue
s.....signal : std_logic
[verbose report available.....]
(:GATE:p) [File: test.vhd]
(:) s <= '0' after 1 ns [File: test.vhd, Line: 20]
```

The following command includes the -verbose option, which causes the output to include resolution function and type conversion function information. This report shows that the port :GATE:q is one of the contributing drivers, and that there is a type conversion function bit\_to\_std through which the value of the port is routed before being assigned to the signal :s. The report also shows that there is a concurrent signal assignment statement contributing as one of the sources to the resolution function.

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
ncsim> drivers s -verbose
s.....signal : std_logic = '0'
'0' <-[resolution function @ieee.std_logic_1164:resolved()]
<src 1>
'0' <- (:GATE) [type conversion function
    bit_to_std(<formal>)]
<formal> connected to port q

:GATE:q....port : inout BIT = '1'
'0' <- (:GATE:p) [File: test.vhd]
<src 2>
'0' <- (:) s <= '0' after 1 ns [File: test.vhd, Line: 20]
```

The following command shows the drivers :gate:q.

```
ncsim> drivers :gate:q
GATE:q.....port : inout BIT = '1'
'0' <- (:GATE:p) [File: test.vhd]
```

The following command includes the **-effective** option, which displays contributions to the effective value of the signal instead of to the driving value.

```
ncsim> drivers :GATE:q -effective
GATE:q.....port : inout BIT = '1'
[verbose report available....]
'0' <- (:GATE:p) [File: test.vhd]
'0' <- (:) s <= '0' after 1 ns [File: test.vhd, Line: 20]
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following command includes the **-verbose** option, which helps you to understand where the effective value of 1 in the previous example comes from.

```
ncsim> drivers :GATE:q -effective -verbose
GATE:q.....port : inout BIT = '1'
'1' <- (:GATE)  [type conversion function std_to_bit(<actual>)]
<actual> connected to signal s

:s.....signal : std_logic = '0'
'0' <- [resolution function @ieee.std_logic_1164:resolved()]
<src 1>
'0' <- (:GATE)  [type conversion function
bit_to_std(<formal>)]
<formal> connected to port q

:GATE:q....port : inout BIT = '1'
'0' <- (:GATE:p) [File: test.vhd]
<src 2>
'0' <- (:) s <= '0' after 1 ns [File: test.vhd, Line: 20]
```

The following command includes the **-future** option, which lists the currently scheduled transactions on each driver.

```
% ncsim -tcl e:a
ncsim: v3.00.(p1): (c) Copyright 1995 - 1999 Cadence Design Systems, Inc.
ncsim> run .5 ns
Ran until 500 PS + 0
ncsim> drivers s -future
s.....signal : std_logic = 'U'
[verbose report available.....]
'0' <- (:GATE:p) [File: test.vhd]
Future Transactions
None Scheduled

'U' <- (:) s <= '0' after 1 ns [File: test.vhd, Line: 20]
Future Transactions
'0' after 1000000.00 fs
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following command shows the output of the `drivers` command when the driver is from a Verilog model.

```
ncsim> drivers -effective i1:a
i1:a.....port : in std_logic = '1'
'1' <- (and2_top.i1) driven by a Verilog model
ncsim> drivers -effective i1:i1:port1
i1:i1:port1...port : in std_logic = '1'
'1' <- (and2_top.i1) port 'a' in and2(and2_bot) [File:
./and2.vhd], driven by a Verilog model
```

## **exit**

The `exit` command terminates simulation and returns control to the operating system.

You also can use the `finish` command to exit a simulation. (See “[finish](#)” on page 549.)

See “[Exiting the Simulation](#)” on page 338 for more information.

### **exit Command Syntax**

```
exit
```

### **exit Command Modifiers and Options**

None.

### **exit Command Examples**

The following command ends the simulation session.

```
ncsim> exit
```

## **finish**

The `finish` command causes the simulator to exit and returns control to the operating system.

This command takes an optional argument that determines what type of information is displayed.

- 0—Prints nothing (same as executing `finish` without an argument).
- 1—Prints the simulation time.
- 2—Prints simulation time and statistics on memory and CPU usage.

See “[Exiting the Simulation](#)” on page 338 for more information.

### **finish Command Syntax**

```
finish [0 | 1 | 2]
```

### **finish Command Modifiers and Options**

None.

### **finish Command Examples**

The following command ends the simulation session.

```
ncsim> finish
```

The following command ends the simulation session and prints the simulation time.

```
ncsim> finish 1
```

```
Simulation complete via $finish(1) at time 0 FS + 0  
%
```

The following command ends the simulation session, prints the simulation time, and displays memory and CPU usage statistics.

```
ncsim> finish 2
```

```
Memory Usage - 7.6M program + 2.1M data = 9.8M total  
CPU Usage - 0.9s system + 2.5s user = 3.4s total (28.5% cpu)  
Simulation complete via $finish(2) at time 500 NS + 0  
%
```

## **fmibkpt**

The **fmibkpt** command performs operations on breakpoints that are coded into C models using the `fmiBreakpoint` call. This command is only available when using the C interface to integrate C models into a VHDL design.

You can:

- Enable FMI breakpoints (`-enable`). FMI breakpoints are all initially disabled.
- Disable FMI breakpoints (`-disable`)
- Display information about FMI breakpoints (`-show`)

### **fmibkpt Command Syntax**

```
fmibkpt {-enable bkpt_number | -disable bkpt_number | -show}
```

### **fmibkpt Command Modifiers and Options**

#### **-enable *bkpt\_number***

Enables the fmi breakpoint with the specified number.

#### **-disable *bkpt\_number***

Disables the fmi breakpoint with the specified number.

#### **-show**

Displays information about fmi breakpoints.

## force

The `force` command sets a specified object to a given value and forces it to retain that value until it is released with a `release` command or until another force is placed on it. (See “[release](#)” on page 592 for details on the `release` command.)

The new value takes effect immediately, and, in the case of Verilog wires and VHDL signals and ports, the new value propagates throughout the hierarchy before the command returns. Releasing a force causes the value to immediately return to the value that would have been there if the force hadn’t been blocking transactions.

The object that is being forced must have write access. An error is printed if it does not. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details on specifying access to simulation objects.

The object cannot be a:

- A Verilog memory
- A Verilog memory element
- A bit-select or part-select of a Verilog register
- A bit-select or part-select of an unexpanded Verilog wire
- A VHDL variable

For Verilog, a force created by the `force` command is identical in behavior to a force created by a Verilog `force` procedural statement. The force can be released by a Verilog `release` statement or replaced by a Verilog `force` statement during subsequent simulation.

The value must be a literal, and the literal is treated as a constant. Even if the literal is generated using [value substitution](#) or Tcl’s `expr` command, the value is considered to be a constant. The forced value will not change if objects used to generate the literal change value during subsequent simulation.

For VHDL, the value specified with the `force` command must match the type and subtype constraints of the VHDL object. Integers, reals, physical types, enumeration types, and strings (including `std_logic_vector` and `bit_vector`) are supported. Records and non-character array values are not supported, but objects of these types can be assigned to by issuing commands for each subelement individually.

Forces created by the `force` command and those created by a Verilog `force` procedural statements are saved if the simulation is saved.

See “[Forcing and Releasing Signal Values](#)” on page 435 for more information.

## force Command Syntax

```
force object_name [=] value
```

## force Command Modifiers and Options

None.

## force Command Examples

### Verilog examples:

The following command forces object *r* to the value '*bx*'. The equal sign is optional.

```
ncsim> force r = 'bx
```

The following command uses value substitution. Object *x* is forced to the current value of *w*.

```
ncsim> force x = #w
```

The following command uses command substitution and value substitution. Object *y* is forced to the result of the Tcl *expr* command, which evaluates the expression `#r[0] & ~#r[1]` using the current value of *r*.

```
ncsim> force y [expr #r[0] & ~#r[1]]
```

The following command shows the error message that is displayed if you run in regression mode and then use the *force* command on an object that does not have write access.

```
ncsim> force clrb 1
```

```
ncsim: *E,RWACRQ: Object does not have read/write access: harddrive.h1.clrb.
```

### VHDL examples:

The following command forces object *:t\_nickel\_out* (*std\_logic*) to 1. The equal sign is optional.

```
ncsim> force :t_nickel_out = '1'
```

The following command forces object *:top:DISPENSE\_tempsig* (*std\_logic*) to 1.

```
ncsim> force :top:DISPENSE_tempsig '1'
```

The following command forces object *:t\_dimes* (*std\_logic\_vector*) to 0.

```
ncsim> force :t_DIMES {"00000000"}
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following command forces object `is_ok` (boolean) to TRUE.

```
ncsim> force :is_ok true
```

The following command forces object `:count` (integer) to 10.

```
ncsim> force :count 10
```

## help

The `help` command displays information about simulator (*ncsim*) commands and options and predefined variable names and values. You can:

- Get help on all commands.
- Get detailed help on a specific command.
- Get help on a command option.
- Display help on all commands that take a specific option.
- Display help for predefined simulation variable names and values.
- Display help for Tcl functions that can be used in expressions.

You also can use the `help` command to display help on standard Tcl commands. Only basic information is provided for these commands. Man pages for Tcl commands, as well as a summary of the Tcl language syntax, can be found on the Web at:

<http://www.elf.org/tcltk-man-html.html>

See “[Getting Help on Simulator Commands](#)” on page 44 for more information.

### help Command Syntax

```
help [help_options] [command | all [command_options]]  
-brief  
-functions [function_name ...]  
-variables [variable_name ...]
```

The special keyword `all` can be used instead of a specific command name. For example, the following command shows full help for all commands:

```
ncsim> help all
```

The following command shows full help for all commands that have the `-enable` option:

```
ncsim> help all -enable
```

## help Command Modifiers and Options

### **-brief**

Displays a list of commands with no description of the commands or options.

### **-functions [function\_name ...]**

Displays help for Tcl functions that can be used in expressions. This includes help on mathematical functions that are predefined in Tcl, as well as special functions that have been added to deal with Verilog values. If no function name is specified, help is displayed for all functions.

### **-variables [variable\_name...]**

Displays a description of the specified predefined simulation variable name(s) and its current value. If no variable name is specified, help is displayed for all predefined variables.

## help Command Examples

The following command displays a list of all *ncsim* commands and Tcl standard commands.

```
ncsim> help
```

The following command displays help for the *probe* command and all its options.

```
ncsim> help probe
```

The following command displays a list of all options to the *probe* command. No description of the command or options is displayed.

```
ncsim> help -brief probe
```

The following command displays a list and description of options that can be used with the *probe* command used with the *-create* modifier.

```
ncsim> help probe -create
```

The following command displays the options that can be used with the *probe* command used with the *-create* modifier. No description of the options is displayed.

```
ncsim> help -brief probe -create
```

The following command displays full help for all *ncsim* commands and basic help for Tcl standard commands.

```
ncsim> help all
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following command displays a simple list of all *ncsim* and Tcl standard commands. No description of the commands or options is displayed.

```
ncsim> help -brief all
```

The following command displays brief help for all commands that have the `-enable` option.

```
ncsim> help -brief all -enable
```

The following command displays help for the `describe` command and for the `-nounit` option of the `time` command.

```
ncsim> help describe time -nounit
```

The following command displays help for the `describe` command, the `-nounit` option of the `time` command, and all commands with the `-enable` option.

```
ncsim> help describe time -nounit all -enable
```

The following command displays help for all predefined simulation variables.

```
ncsim> help -variables
```

The following command displays a description of the predefined simulation variable `time_scale`.

```
ncsim> help -variables time_scale
```

```
time_scale = NS.....Timescale of the current debug scope (read only)
```

The following command displays help for all Tcl functions that can be used in expressions.

```
ncsim> help -functions
```

## history

The `history` command is a built-in Tcl command that lets you reexecute commands without having to retype them. You also can use the `history` command to modify old commands—for example, to fix typographical errors.

The `history` command is similar to the UNIX `history` command, but with different syntax in some cases. You can:

- Modify the number of commands retained by the history mechanism (`keep`). By default, the history mechanism keeps track of the last twenty commands.
- Reexecute commands by giving their event number (`redo`).
- Modify parts of a previous command before reexecuting it (`substitute`).

You can also use the `!` command to reexecute commands.

See the Tcl documentation for complete details on the `history` command.

### history Command Syntax

```
history
  keep n
  redo event_number
  substitute old new event_number
```

**Note:** This list of options for the `history` command is partial. See the Tcl documentation for details on all options.

### history Command Options

#### **redo [*event\_number*]**

Reexecutes the command specified by the `event_number` argument. The argument can be:

- A positive number  
Reexecutes the command with that number.

■ A negative number

Reexecutes a command relative to the current command. For example, `-1` reexecutes the last command, `-2` reexecutes the one before that, etc.

■ A string

Reexecutes the command that matches the string. The string matches the command if it is the same as the first characters of the command or it meets Tcl's rules for string matching.

If no argument is specified, `redo` reexecutes the most recent command.

**keep *n***

Changes the number of commands retained by the history mechanism. The default is the 20 most recent commands.

**substitute *old new event\_number***

Modifies the old command before executing it.

## history Command Examples

The `history` command is a built-in Tcl command with many features and options. The examples here illustrate only the most commonly used options.

Assume you have entered the following seven commands:

```
ncsim> database -open waves -into waves1.shm  
ncsim> database -show  
ncsim> stop -create -line 10  
ncsim> stop -show  
ncsim> run  
ncsim> run 100 ns  
ncsim> value data
```

The following sequence of commands illustrates some of the most useful features of the `history` command:

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
# Get a list of interactive commands.  
ncsim> history  
1 database -open waves -into waves1.shm  
2 database -show  
3 stop -create -line 10  
4 stop -show  
5 run  
6 run 100 ns  
7 value data  
8 history  
  
# Reexecute the second from the last command (value data). Same as !-2.  
ncsim> history redo -2  
4'b0000  
  
# Reexecute command number 6 (run 100 ns). Same as !6.  
ncsim> history redo 6  
Ran until 200 NS + 0  
  
# Reexecute the last command matching the string dat. Same as !dat.  
ncsim> history redo dat  
waves Enabled (file: waves1.shm) (SHM)  
  
# Reexecute command number 3, changing 10 to 11.  
ncsim> history substitute 10 11 3  
Created stop 2
```

## input

The `input` command pushes the Tcl commands in the specified script file(s) onto the standard input queue so that *ncsim* reads and executes them at the next `ncsim>` Tcl prompt. The simulator executes the commands that are contained in the script file(s) as if you had typed them at the prompt.

You can also execute commands in a script file by using the `-input` option when you invoke the simulator. Like the `input` command, the `-input` option takes a script file as an argument and queues the commands in the file so that the simulator executes them when it issues its first prompt.

You can specify more than one script file with the `input` command. If you specify more than one file, the files are executed in the order that they appear on the command line.

A script file that you specify with the `input` command can contain other `input` commands. When *ncsim* executes an embedded `input` command, it pushes the contents of the embedded `input` command file to the front of the input queue so that the commands are executed before the remaining commands from the original script file. That is, the simulator executes the commands in an embedded script file as if you included those commands in the original script file.

You can also execute Tcl commands in a script file by using the `source` command. However, because the commands that are pushed onto the input queue by the `input` command are not read until a prompt is issued, `input` commands that are embedded in a script file that is executed with the `source` command do not take effect until after the `source` command finishes and another prompt is issued. *ncsim* pushes the scripts from all of the `input` commands that it encounters when executing the `source` command so that they are executed in the order that they were encountered.

The `input` command also differs from the `source` command in the following ways:

- With the `source` command, execution of the commands in the script stops if a command generates an error. With the `input` command, the contents of the file are read in place of standard input at the next Tcl prompt, as if you had typed the commands at the command-line prompt. This means that errors do not stop the execution of commands in the script.
- The `source` command displays the output of only the last command in the file. The `input` command, on the other hand, echoes commands to the screen as they are executed, along with any command output or error messages.

## **input Command Syntax**

```
input script_file [script_file ...]
```

## **input Command Modifiers and Options**

None.

## **input Command Examples**

In the following example, the snapshot is loaded into *ncsim* and then an *input* command is issued to execute the commands in the file *tcl.inpl*. This is the same as using the *-input* option on the *ncsim* command line when you invoke the simulator. The file *tcl.inpl* contains the following Tcl commands:

```
stop -create -line 27
run
value data
run 100 ns
value data
run 100 ns
value data
run

% ncsim -tcl worklib.harddrive
ncsim: v3.00.(s5): (c) Copyright 1995 - 2000 Cadence Design Systems, Inc.
Loading snapshot worklib.harddrive:module ..... Done
ncsim> input tcl.inpl
ncsim> stop -create -line 27
Created stop 1
ncsim> run
0 FS + 0 (stop 1: ./harddrive.v:27)
./harddrive.v:27      repeat (2)
ncsim> value data
4'hx
ncsim> run 100 ns
Ran until 100 NS + 0
ncsim> value data
4'h0
ncsim> run 100 ns
Ran until 200 NS + 0
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
ncsim> value data
4'h1
ncsim>
```

In the following example, an `input` command is issued to execute the commands in the file `tcl.inp2`. Notice that, unlike the `source` command, errors do not stop the execution of commands in the script. The simulator generates an error message for the second `probe` command, and then the following command is executed.

The file `tcl.inp2` contains the following Tcl commands:

```
database -open -shm waves -default
probe -create -database waves clk
probe -create -database waves nonexistent_signal
probe -create -database waves data

% ncsim -tcl worklib.hardrive
ncsim: v3.00.(s5): (c) Copyright 1995 - 2000 Cadence Design Systems, Inc.
Loading snapshot worklib.hardrive:module ..... Done
ncsim> run 50 ns
Ran until 50 NS + 0
ncsim> input tcl.inp2
ncsim> database -open -shm waves -default
SST2 Database Write API -- DWAPI Version 2.5s3 -- 01/14/2000
Copyright 1997-2000 Cadence Design Systems, Inc.
```

```
Created default SHM database waves
ncsim> probe -create -database waves clk
Created probe 1
ncsim> probe -create -database waves nonexistent_signal
ncsim: *E,PNOOBJ: Path element could not be found: non_existent_signal.
ncsim> probe -create -database waves data
Created probe 2
ncsim>
```

## memory

The `memory` command loads VHDL memory from a memory file or dumps VHDL memory to a memory file. The command has two modifiers: `-load` and `-dump`.

- Use the `-load` option to load VHDL memory from a memory file. The `memory -load` command reads the contents of a memory image file into a VHDL array object.
- Use the `-dump` option to dump VHDL memory to a memory file. The `memory -dump` command writes out the contents of a VHDL array object into a specified memory image file.

### VHDL Array Object

The VHDL array object must be a one-dimensional array, and its index subtype must be an integer type. The element type of the variable must also be a one-dimensional array. The type of the array elements must be an enumeration type that contains the literals 0 and 1, and that can contain any of the enumeration literals in `std_logic`. Here is an example of a VHDL array that satisfies these conditions:

```
type MVL4 is (X, 0, 1, Z);
type MVL4_VECTOR is array (NATURAL range <>) of MVL4;
type MEMTYPE is array (NATURAL range <>) of MVL4_VECTOR (31 downto 0);
variable MEM: MEMTYPE (0 to 1023);
```

The VHDL array object must have write access. Use the `ncelab -access +w` option (`ncverilog +access+w`) when you elaborate the design.

### Memory Image File

A memory image file contains:

- Directives for specifying the address format and the data format.
  - `$ADDRESSFMT address_format`
  - `$DATAFMT data_format`

*address\_format* and *data\_format* can be H (Hex), B (Binary), or O (Octal).

The memory image file must have a `$ADDRESSFMT` directive as the first non-comment line.

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

You can also specify a default value for unspecified addresses by using the `$DEFAULTVALUE default_value` directive, where `default_value` can be 0, 1, X (unknown), or any enumeration literal in `std_logic`.

- One or more lines that specify the address(es) to be filled and the data to be loaded. You can specify the address and data in hexadecimal, octal, or binary format.

The format is `start_address[:end_address]/data`. For example:

`0:3/0` (sets address 0 to 3 to value 0)

`4/a;b;c;d;e` (sets address 4 to 8 to values a, b, c, d, and e). Note the semi-colons that separate the data values.

The following is an example memory image file. Comments in the file begin with a pound (#) sign.

```
# Address format: H(Hex) or B(Binary), or O(Octal).
$ADDRESSFMT H
# Data format: H(Hex) or B(Binary), or O(Octal).
$DATAFMT H
# Default value for unspecified addresses: 0, 1, or X(Unknown).
# If default value is omitted, only the memory addresses that are
# specified are filled. Other addresses remain unchanged.
$DEFAULTVALUE X
# The following line sets the address 0 to value 0.
0/0
# The following line sets address 1 to 5 to values: a, b, c, d, and e.
1/a;b;c;d;e
# The following line sets the address range a:1f(in Hex) to value f # (in Hex).
a:1f/f
# Values for address 6 to 9 are not specified, and will be set to X, # the default
value.
```

The `memory -load` command loads the VHDL array based on the address that you specify in the memory image file.

If the memory image file has fewer elements than the VHDL array, the corresponding elements (addresses) of the VHDL array are changed. The rest of the VHDL array elements are either set to the default value that you specify with the `$DEFAULTVALUE` directive or they remain unchanged (if no `$DEFAULTVALUE` directive is specified).

If the memory image file has an address that does not exist in the VHDL array, the simulator issues a warning message and ignores such elements in the memory image file.

The `memory -load` command fills each element of the VHDL array from LSB to MSB. If the width of the VHDL array element is smaller than the width of the memory image file element, the most significant (left-most) bits of the memory image file element are lost. If the memory image file has fewer elements than the VHDL array, the corresponding bits of the element are changed. The rest of the VHDL bits are either set to the default value specified with the `$DEFAULTVALUE` directive or remain unchanged (if no `$DEFAULTVALUE` directive is specified).

Use the `-dump` option to dump VHDL memory to a specified memory file. The restrictions on the array object are the same as those described in “[VHDL Array Object](#)” on page 563.

With the `memory -dump` command, you can specify an output format (hexadecimal, octal, or binary) and the start and end addresses of the memory being dumped.

You can read back the memory image file produced with `memory -dump` to load VHDL memory. This file does not contain a `$DEFAULTVALUE` directive. The simulator dumps only the addresses that you specify, so that a load in the same simulation session cannot destroy other memory locations.

## **memory Command Syntax**

```
memory -load vhdl_array_object [-file] mem_filename
```

```
memory -dump [output_format] [-start start_addr] [-end end_addr]  
          vhdl_array_object [-file] mem_filename
```

## **memory Command Modifiers and Options**

### **Loading VHDL Memory**

#### **`-load vhdl_array_object [-file] mem_filename`**

Reads the contents of the specified memory image file into the VHDL array object specified by `vhdl_array_object`.

You must specify the `vhdl_array_object` argument before the `mem_filename` argument unless you use the `-file` option. If you want to specify the filename first, you must use the `-file` option. For example:

```
ncsim> memory -load -file ram.mem :c_ram:u1:m
```

## Dumping VHDL Memory

**-dump [*output\_format*] [-start *start\_addr*] [-end *end\_addr*]  
*vhdl\_array\_object* [-file] *mem\_filename***

Dumps the contents of the VHDL array object specified by *vhdl\_array\_object* to the specified memory file.

The output format options are:

- %h—hexadecimal (This is the default)
- %o—octal
- %b—binary

Use the -start and -end options to specify the start and end addresses of the memory being dumped. The arguments to these options are integers, which you can specify in decimal, hexadecimal, or octal notation.

You must specify the *vhdl\_array\_object* argument before the *mem\_filename* argument unless you use the -file option. If you want to specify the filename first, you must use the -file option.

## memory Command Examples

An example of a VHDL design with a memory component is shown below. DLATRAM is defined in the package IEEE.std\_logic\_components.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_components.all;

entity ram_top is
    port (
        DATAin  : in STD_LOGIC_VECTOR(3 downto 0);
        DATAout : out STD_LOGIC_VECTOR(3 downto 0);
        ADDR    : in STD_LOGIC_VECTOR(4 downto 0);
        WE      : in STD_LOGIC;
        RE      : in STD_LOGIC
    );
end ram_top;
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
architecture A of ram_top is
begin
  C_RAM: DLATRAM
    generic map ( 4, 5, 6 ns, 9 ns)
    port map (DATAin, DATAout, ADDR, WE, RE);
end A;
```

The memory image file to be loaded in this example (`c_ram.mem`) is as follows:

```
$ADDRESSFMT H
$DATAFMT H
$DEFAULTVALUE X
0:0/0
1/a;b;c;d;e
a:1f/f
```

The following command loads the contents of the memory file (`c_ram.mem`) into the VHDL memory.

```
ncsim> memory -load :C_RAM:p:m c_ram.mem
```

The following command dumps the contents of addresses 0 to 5 of the VHDL memory array to a memory file called `memfile.mem`.

```
ncsim> memory -dump :C_RAM:p:m -start 0 -end 5 memfile.mem
```

## omi

The `omi` command lets you display information about model managers and instances controlled by model managers. It also lets you pass OMI model manager run-time commands to model managers that support this capability. Some model managers provide special capabilities that enhance the usability of the models under its control. For example, a model manager might let you load the contents of a memory viewport from a file or let you dynamically control the collection of simulation data.

Use the `omi` command to:

- Display information on the model managers and model instances for the current simulation session (`-list`).
- Send commands to model managers and model instances (`-send`).

See “[The Open Model Interface \(OMI\)](#)” on page 952 for details on importing OMI models.

### omi Command Syntax

```
omi
  -list
    [-all]
    [-manager]
    [-instance [model_manager]]
  -send
    [-all] command
    [-manager model_manager command]
    [-instance instance_name command]
```

## **omi Command Modifiers and Options**

The `omi` command has two modifiers: `-list`, which is used to display information, and `-send`, which is used to issue commands.

### **Displaying Information**

#### **-list -all**

Displays a list of all model managers and the instances managed by each model manager. The model manager aliases are also listed.

#### **-list -manager**

Displays a list of the model managers. The list includes the model manager aliases and the corresponding names given by the model managers.

#### **-list -instance [*model\_manager*]**

Displays a list of all OMI instances. If you include a model manager alias, only the instances controlled by that model manager are listed.

### **Issuing Commands**

#### **-send [-all] *command***

Sends the command to all model managers.

#### **-send -manager *model\_manager* *command***

Sends the command to the specified model manager.

#### **-send -instance *instance\_name* *command***

Sends the command to the specified model instance.

## omi Command Examples

The following command displays a list of the OMI model managers and instances information for the current simulation session.

```
ncsim> omi -list -all
```

The following command displays a list of the model managers. The list includes model manager aliases and the corresponding names given by the model managers.

```
ncsim> omi -list -manager
```

The following command displays a list of all OMI instances.

```
ncsim> omi -list -instance
```

The following command displays a list of all OMI instances controlled by the model manager mm:1.

```
ncsim> omi -list -instance mm:1
```

The following command sends the specified command to all model managers for all model instances.

```
ncsim> omi -send -all "dump mem"
```

The following command sends the command to model instances controlled by the specified model manager.

```
ncsim> omi -send -manager mm:1 "dump mem"
```

The following command sends the command to the specified model instance only.

```
ncsim> omi -send -instance top.p1 "loadmem mem ./mem_file"
```

## probe

The `probe` command lets you control the values that you save to an SHM, Value Change Dump (VCD), or Extended Value Change Dump (EVCD) database. You can:

- Create probes by using the optional `-create` modifier. You can probe objects to the following kinds of databases:
  - SHM (for Verilog, VHDL, or mixed-language)
  - VCD (for Verilog, VHDL, or mixed-language)
  - EVCD (for VHDL only).

For Verilog you must use the `$dumpports` system task. See “[Generating an Extended Value Change Dump \(EVCD\) File](#)” on page 463.
- Delete probes (`probe -delete`).
- Disable probes (`probe -disable`).
- Enable previously disabled probes (`probe -enable`).
- Display information about probes (`probe -show`).

### Probing Verilog Objects

You can use the `probe` command to probe Verilog objects to an SHM or VCD database. You must use the `$dumpports` system task to probe objects to an EVCD database. See “[Generating an Extended Value Change Dump \(EVCD\) File](#)” on page 463 for details.

You can only probe objects that have read access. If you specify an object as an argument to the `probe` command, and that object does not have read access, the simulator prints an error message. If you specify a scope as an argument to the `probe` command, objects within that scope that do not have read access are excluded from the probe, and the simulator prints a warning message. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details on specifying access to simulation objects.

See Section 15 of the *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language* (IEEE Std 1364-1995) for a full description of the syntax and format of the VCD file.

## Probing VHDL Objects

For VHDL, you cannot probe objects that are declared inside a subprogram. You can probe all VHDL signals, ports, and variables that are not declared inside subprograms to an SHM database unless their type falls into one of the following categories:

- Non-standard integer types whose bounds require more than 32 bits to represent
- Physical types
- Access and file types
- Any composite type that contains one of the above types

For VHDL, the syntax and format of the VCD file is identical to Verilog. You can dump signals, ports, and variables of type `std_ulogic`, `bit`, `integer`, `real`, or any user-defined type that is a subset of the above. Signals and ports in VHDL map to wires in Verilog, and variables map to registers.

The values that can be dumped to a VCD file are 0, 1, Z, and X. Other values are mapped as follows:

- U -> X
- W -> X
- L -> 0
- H -> 1
- - -> X

**Note:** In the current release, VHDL signals and variables that correspond to records are not dumped to a VCD file.

You can also create a VCD database and probe VHDL objects to the database by using the `call` command to call predefined CFC routines, which are part of the NC-Verilog simulator C interface. This feature has been retained for backwards compatibility. The recommended method of generating a VCD file is to open a database with the `database -open -vcd` command and to probe objects to the database with the `probe -vcd` command. See “[call](#)” on page 504 for details on the `call` command. See the appendix called “Generating a VCD File Using CFC Routines” in the *NC-VHDL Simulator Help* for details on this alternate method of generating a VCD file.

For an EVCD database, you can probe only the primary ports of a component instance(s). The simulator monitors the ports for both value and drive level, and generates an output file that contains the value, direction, and strength of the primary ports of the component instance(s). See the section called “Generating an Extended Value Change Dump (EVCD)

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

File” in the chapter called “Debugging Your Design” *NC-VHDL Simulator Help* for more information on EVCD databases.

## probe Command Syntax

```
probe
[-create] [{object | scope_name}...]
[ { -shm | -vcd | -evcd [{splitio | simple}] | -database dbase_name } ]

If you do not specify an object or scope_name argument, you must use one of the following options: -inputs,-outputs, -ports, -all.
You must specify the database type (-shm, -vcd, -evcd) or the database name if more than one database is open.

[-all]
[-depth {n | all | to_cells}]
[-inputs]
[-name probe_name]
[-outputs]
[-ports]
[-screen [-format format_string] [-redirect filename] objects]
[-variables]
[-waveform]

-delete probe_name [probe_name ...]

-disable probe_name [probe_name ...]

-enable probe_name [probe_name ...]

-save [filename]

-show [probe_name ...] [-database dbase_name]
```

You can use wildcard characters in the argument to a probe -create command if the argument is a Verilog or VHDL object name.

- The asterisk ( \* ) matches any number of characters.
- The question mark ( ? ) matches any one character.

You cannot use wildcard characters in scope specifiers or inside escaped names.

The argument to `-delete`, `-disable`, `-enable`, or `-show` is a probe name or a list of probe names. You can use the two wildcard characters (`*` and `?`) in the probe name argument.

See “[Using Wildcards Characters in Tcl Commands](#)” on page 495 for more information on using wildcards.

## probe Command Modifiers and Options

### Creating a Probe

```
-create [ {object | scope_name} ... ]  
[ { -shm | -vcd | -evcd [{splitio | simple}] } | -database dbase_name } ]
```

Places the values of the specified objects in a database. The `-create` modifier is optional.

Simulation objects must have read access in order to be probed.

The `-create` modifier can be followed by an argument that specifies:

- The object(s) to be traced
- The scope(s) to be traced
- A combination of object(s) and scope(s) to be traced

If you do not specify an argument, the current debug scope is assumed, but you must include an option that specifies which objects to include in the trace (`-all`, `-inputs`, `-outputs`, or `-ports`).

If more than one database is open, you must include an option to specify the database into which you want to dump values. Use one of the following options:

- `-database dbase_name`  
Send the probe to the specified database. The database must already exist.
- `-shm`  
Send the probe to the default SHM database. If no default database is open, the simulator opens a default database called `ncsim.shm` is opened.

■ **-vcd**

**Note:** For Verilog, you cannot probe signals to an EVCD database by using Tcl commands. For Verilog, you must use the `$dumpports` system task to generate an EVCD file.

Send the probe to the default VCD database. If no default database is open, the simulator opens a default database called `ncsim.vcd` is opened.

■ **-evcd**

Send the probe to the default EVCD database. If no default database is open, the simulator opens a default database called `ncsim.evcd` is opened.

**-all**

Specifies that all of the declared objects within a scope, except for VHDL variables, are to be included in the probe. This applies to:

- The scope(s) named in the argument
- The current debug scope, if no scope or object is named in an argument
- The subscopes that you specify with the [-depth](#) option

Use `-all -variables` to include VHDL variables in the probe.

For EVCD, the `-all` option (or the `-ports` option) probes all of the declared primary ports within a scope.

**-database *dbase\_name***

Saves the probe into the specified database.

If more than one database is open, you must specify the database into which you want to dump values. You can do this by specifying a database name with the `-database` option. The *dbase\_name* argument is the logical name of the database in which you want to save the probe. This database must be open. The `-database` option does not create a database for you.

You can also specify the database into which you want to dump values with the `-shm`, `-vcd`, or `-evcd` option. These options specify that you want to save the probe to the default SHM, VCD, or EVCD database, respectively.

If only one database is open, you do not need to specify a database.

### **-depth { n | all | to\_cells }**

Specifies how many scope levels to descend when searching for objects to probe if you have specified a scope. You must specify one of the following arguments:

- *n*

Descend the specified number of scopes. For example, `-depth 1` means include only the given scope, `-depth 2` means include the given scope and its subscopes, and so on. The default is 1.

- *all*

Include all scopes in the hierarchy below the specified scope(s).

- *to\_cells*

Include all scopes in the hierarchy below the specified scope(s), but stop at cells (modules with ``celldefine`).

You cannot use this argument if you are probing to an EVCD database because you cannot use Tcl commands to probe Verilog signals to an EVCD database.

### **-evcd [ {splitio | simple} ]**

**Note:** You cannot generate an EVCD file for Verilog by using Tcl commands. For Verilog, you must use the `$dumpsupports` system task to generate an EVCD file.

Sends the probe to the default EVCD database. The simulator opens a default database named `ncsim.evcd` if one does not already exist. The associated file is `ncsim.evcd`. The file is placed in the current working directory.

If you have opened an EVCD database with the `database` command, and if this is the only database of any kind that is open, a `probe` command will automatically use that database. It is not necessary to use the `-evcd` option or the `-database` option to specify the database.

By default, the simulator dumps final value changes in extended format for both vector and scalar ports. For vector ports, the simulator dumps value changes for vector elements for which there was an event on the driver(s) of that particular element one at a time.

There are two arguments that you can use with the `-evcd` option. If you do not specify an argument, the simulator dumps in the default mode described in the previous paragraph.

- **splitio**

Dump the value of the input port if any driver into the entity changes value, and dump the value of the output port if any driver within the entity changes value. If both inside and outside drivers change, dump two messages: one corresponding to the input and one corresponding to the output.

- **simple**

Dump resolved signal values on every transaction on the signal connected to a given primary port in simple 0X1Z format.

#### **-inputs**

Specifies that all inputs within a scope are to be included in the probe. This applies to:

- The current debug scope (if no scope(s) or object(s) is named in an argument)
- The scope(s) named in the argument
- Subscopes that you specify with the [-depth](#) option

#### **-name *probe\_name***

Specifies a user-defined name for the probe. You can then use the name that you assign to a probe with the `-disable`, `-enable`, `-delete`, and `-show` modifiers.

If you do not use `-name` to name your probes, the simulator gives every probe that you create a sequential number.

#### **-outputs**

Specifies that all outputs within a scope are to be included in the probe. This applies to:

- The current debug scope (if no scope(s) or object(s) is named in an argument)
- The scope(s) named in the argument
- Subscopes that you specify with the [-depth](#) option

### **-ports**

Specifies that all ports within a scope are to be included in the probe. This applies to:

- The current debug scope (if no scope(s) or object(s) is named in an argument)
- The scope(s) named in the argument
- Subscopes that you specify with the [-depth](#) option

For EVCD, you can use `-ports` or `-all` to include all of the declared ports within a scope in the probe.

### **-screen [-format *format\_string*] [-redirect *filename*] *objects***

Monitors the value changes on the specified object(s) and displays the values on the screen.

You cannot use this option with any other `probe` command-line option.

Include the `-redirect` option to redirect the output to a file. For example:

```
probe -screen -redirect myfile sum
```

Use the `-format` option to specify the output format. The *format\_string* argument can contain both text and format specifiers. Variables and objects are paired sequentially with specifiers.

Valid formats are:

- `%b`—Binary format. The argument must be an object name that is either a scalar object or a logic vector.
- `%d`—Decimal format. The argument must be an object name whose type is integer, physical, or enumeration.
- `%f`—Real number in floating-point notation. The argument must be an object whose base type is real.
- `%o`—Unsigned octal object. The argument must be a scalar object or a logic vector.
- `%s`—Substitute. The argument is substituted on an as-is basis.
- `%x`—Unsigned hex object. The argument is an object name that is either a scalar object or a logic vector.
- `%v`—Default value format. The argument must be a signal or a variable name. The value of the object is formatted appropriately, according to its type.

- \t—Inserts a horizontal tab.
- \n—Inserts a carriage return.
- \c—Used as the last character, this suppresses the default line feed.
- %C—Prints the cycle count.
- %D—Prints the current delta cycle count.
- %T—Prints the current simulation time.

#### **-shm**

Sends the probe to the default SHM database. The simulator opens a default database named `ncsim.shm` if one does not already exist. The associated file is `ncsim.shm`. The simulator places the file in the current working directory.

If you have opened an SHM database with the `database` command, and if this is the only database of any kind that is open, a `probe` command will automatically use that database. It is not necessary to use the `-shm` option or the `-database` option to specify the database.

#### **-variables**

Includes VHDL variables when you probe all objects in a scope. You must use the `-all` option with `-variables`.

Because you can probe only the primary ports of a component instance(s) to an EVCD database, you cannot use the `-variables` option if you are probing to an EVCD database.

#### **-vcd**

Sends the probe to the default VCD database. The simulator opens a default database named `ncsim.vcd` if one does not already exist. The associated file is `ncsim.vcd`. The simulator places the file in the current working directory.

If you have opened a VCD database with the `database` command, and if this is the only database of any kind that is open, a `probe` command will automatically use that database. It is not necessary to use the `-vcd` option or the `-database` option to specify the database.

### **-waveform**

Causes the objects in the probe to be added to the SimVision waveform display if the simulator is running in GUI mode. If the waveform viewer is not running, it is invoked and the objects are then added to the display.

This option has no effect if the simulator is not in GUI mode.

## **Deleting a Probe**

### **-delete *probe\_name* [*probe\_name* ...]**

Deletes the probe(s) specified by the argument.

The argument to `-delete` is a probe name or a list of probe names. You can use the two wildcard characters (`*` and `?`) in the probe name argument.

- The asterisk (`*`) matches any number of characters
- The question mark (`?`) matches any one character

You can delete SHM probes at any time.

You can only delete VCD or EVCD probes at the time the VCD or EVCD database is created. Once the simulation is advanced, the VCD or EVCD header is written to the file and no modifications to the probes are possible.

## **Disabling a Probe**

### **-disable *probe\_name* [*probe\_name* ...]**

Disables the probe(s) specified by the argument. While the probe is disabled, values for the objects in that probe are not written to the database. To resume probing, use the `-enable` modifier.

The argument to `-disable` is a probe name or a list of probe names. You can use the two wildcard characters (`*` and `?`) in the probe name argument.

- The asterisk (`*`) matches any number of characters
- The question mark (`?`) matches any one character

You can disable SHM probes individually at any time.

You cannot disable VCD or EVCD probes individually. Use `database -disable` to disable all VCD or EVCD probes. (See “[database](#)” on page 517.)

## Enabling a Probe

### **-enable *probe\_name* [*probe\_name* ...]**

Causes a previously disabled probe(s) to be resumed. As soon as the probe resumes, all objects in the probe have their values written to the database.

The argument to `-enable` is a probe name or a list of probe names. You can use the two wildcard characters (\*) and (?) in the probe name argument.

- The asterisk (\*) matches any number of characters
- The question mark (?) matches any one character

## Saving a Script to Recreate Probes

### **-save [*filename*]**

Creates a Tcl script that you can execute to recreate the current databases and probes. If you do not specify a *filename* argument, the script is printed to the screen.

## Displaying Information About Probes

### **-show [*probe\_name* ...] [-database *dbase\_name*]**

Provides information about the probe(s) specified in the argument. The argument to `-show` is a probe name or a list of probe names. You can use the two wildcard characters (\*) and (?) in the probe name argument.

- The asterisk (\*) matches any number of characters
- The question mark (?) matches any one character

If you do not include an argument, the simulator displays information on all probes.

Use the `-database` option to print information on probes in a specified database.

## probe Command Examples

In the following example, the `database` command opens a default SHM database called `waves`. The `probe` command creates a probe on all objects in the current debug scope. Data is sent to the default SHM database. The `-create` modifier is not required. The `-all` option (or `-inputs`, `-outputs`, or `-ports`) is required because no `object` or `scope_name` argument is specified.

In this example, only one database, a default SHM database, is opened. The `-shm` option on the `probe` command is, therefore, optional.

```
ncsim> database -open -shm waves -default  
SST2 Database Write API -- DWAPI Version 2.5s3 -- 01/14/2000  
Copyright 1997-2000 Cadence Design Systems, Inc.
```

```
Created default SHM database waves  
ncsim> probe -create -shm -all  
Created probe 1
```

The following command creates a probe on all objects in the current debug scope. In this example, no default SHM database exists, so the simulator creates a default database called `ncsim.shm`.

```
ncsim> probe -create -shm -all  
SST2 Database Write API -- DWAPI Version 2.5s3 -- 01/14/2000  
Copyright 1997-2000 Cadence Design Systems, Inc.
```

```
Created default SHM database ncsim.shm  
Created probe 1
```

The following command creates a probe on all signals in `top`. Data is sent to the database `waves2`. This database must already exist.

```
ncsim> probe -database waves2 top
```

The following command creates a probe on all objects in scope `top.u1` and sends data to the default SHM database (creating one called `ncsim.shm`, if a default database does not exist).

```
ncsim> probe -shm top.u1
```

The following command creates probes on all objects in the current scope that have names beginning with `rst` and ending with `p5`.

```
ncsim> probe -shm rst*p5
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following command creates probes on all objects in the scope `top` that have two letters and that have names that start with the letter `c`.

```
ncsim> probe -shm top.c?
```

The following command creates a probe on all objects in scopes `top.u1` and `top.u2`.

```
ncsim> probe -shm top.u1 top.u2
```

The following command creates a probe on the signal `sum` in the current debug scope and sends data to the default SHM database.

```
ncsim> probe -shm sum
```

The following command creates a probe on `sum` and `c_out` in the current debug scope and sends data to the default SHM database.

```
ncsim> probe -shm sum c_out
```

The following command creates a probe on `sum` in scope `u1`, sending data to the default SHM database.

```
ncsim> probe -shm u1.sum
```

The following command creates a probe on all ports in scope `u1`.

```
ncsim> probe -shm u1 -ports
```

The following command creates a probe on all ports in scope `u1` and its subscopes.

```
ncsim> probe -shm u1 -ports -depth 2
```

The following command creates a probe on all ports in scope `u1` and all scopes below `u1`.

```
ncsim> probe -shm u1 -ports -depth all
```

The following command creates a probe on all ports in scope `u1` and all scopes below `u1`, stopping at modules with a `celldefine directive.

```
ncsim> probe -shm u1 -ports -depth to_cells
```

The following command creates a probe called `peek`.

```
ncsim> probe -shm sum -name peek
```

In the following example, the `database` command opens a default VCD database called `test_vcd`. The `probe` command creates a probe on all ports in the scope `counter`. Data is sent to the default VCD database.

```
ncsim> database -open -vcd test_vcd -default
```

```
Created default VCD database test_vcd
```

```
ncsim> probe -create -vcd top.counter -ports
```

```
Created probe 1
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following `probe -vcd` command creates a probe on all objects in the current debug scope. In this example, no default VCD database exists, so the simulator creates a default database called `ncsim.vcd`.

```
ncsim> probe -create -vcd -all
Created default VCD database ncsim.vcd
Created probe 1
```

In the following example, the `database` command opens a default EVCD database called `test_evcd`. The `probe` command creates a probe on all primary ports in the scope `:u1`. Data is sent to the default EVCD database.

```
ncsim> database -open -evcd test_evcd -default
Created default EVCD database test_evcd
ncsim> probe -create -evcd -all :u1
Created probe 1
```

The following `probe -evcd` command creates a probe on all primary ports in the current debug scope. In this example, no default EVCD database exists, so the simulator creates a default database called `ncsim.evcd`.

```
ncsim> probe -create -evcd -all
Created default EVCD database ncsim.evcd
Created probe 1
```

The following command monitors value changes on signals `clock` and `count`. When either of these signals changes value, the simulator displays output on the screen.

```
ncsim> probe -screen clock count
Created probe 1
ncsim> run 10 ns
Time: 5 NS: board.clock = 1'h1 : board.count = 4'hx
Ran until 10 NS + 0
```

In the following command, the `-format` option is included to format the output of `probe -screen`.

```
ncsim> probe -screen -format "clock = %d \n count = %b" clock count
Created probe 1
ncsim> run 10 ns
clock = 1'd1
count = 4'bxxxx
Ran until 10 NS + 0
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following example illustrates the simulator output when you use `probe -screen` to monitor signal value changes, and then disable the probe at some later time.

```
ncsim> probe -screen clock count
Created probe 1
ncsim> run 10 ns
Time: 5 NS: board.clock = 1'h1: board.count = 4'hx

Ran until 10 NS + 0
ncsim> probe -disable 1
ncsim> run 10 ns
Time: 10 NS: board.clock = <disabled> : board.count = <disabled>

Ran until 20 NS + 0
```

The following command displays the state of all probes.

```
ncsim> probe -show
```

The following command displays the state of the probe called `peek`.

```
ncsim> probe -show peek
```

The following command disables the probe called `peek`.

```
ncsim> probe -disable peek
```

The following command enables the probe called `peek`, which was disabled in the previous command.

```
ncsim> probe -enable peek
```

The following sequence of commands illustrates the use of wildcard characters in probe name arguments. Two probes called `peek1` and `peek2` are created. Both probes are then disabled using the `*` wildcard character. Both probes are then deleted using the `?` wildcard character.

```
ncsim> probe -database waves clock -name peek1
Created probe peek1
ncsim> probe -database waves count -name peek2
Created probe peek2
ncsim> probe -show
peek1    Enabled          board.clock (database: waves) -shm
peek2    Enabled          board.count (database: waves) -shm
ncsim> probe -disable pe*
ncsim> probe -show
peek1    Disabled         board.clock (database: waves) -shm
peek2    Disabled         board.count (database: waves) -shm
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
ncsim> probe -delete peek?  
ncsim> probe -show  
No probes set  
ncsim>
```

The following command shows the error message that is displayed if you run in the default “regression” mode (no read, write, or connectivity access to simulation objects) and then probe an object that does not have read access.

```
ncsim> probe -shm d  
ncsim: *E,RDACRQ: Object does not have read access: harddrive.h1.d.
```

## process

The `process` command displays information about the processes that are currently executing or that are scheduled to execute at the current simulation time. You can use this command to display the processes in VHDL, in Verilog, or in mixed VHDL/Verilog designs.

For VHDL, the following constructs qualify as processes:

- VHDL process statements
- Postponed processes
- Concurrent statements (implicit process)
- Foreign processes (FMI)

For Verilog, the following constructs qualify as processes:

- `initial` and `always` statement blocks
- Continuous assignments
- Implicit continuous assignments (port connections)
- Non-blocking assignments
- `$monitor` statements

**Note:** You must compile the source code with the `-linedebug` option (non-optimization mode) in order for the `process` command to display correct information.

## process Command Syntax

```
process
  [-all] [count]
  [-current] (Default if no option is specified)
  [-eot] [count]
  [-next] [count]
```

## process Command Modifiers and Options

### **-all [count]**

If you do not specify a *count* argument, process -all lists all of the processes that are scheduled to execute at the current simulation time. This option shows the process that is currently executing (-current), all scheduled processes (-next), and all processes that are scheduled to execute at the end of the current simulation time (-eot).

If you specify a *count* argument, the number of processes in each of these categories is limited to the specified number. For example:

```
ncsim> process -all 2
```

### **-current**

Displays the process that is currently executing.

This option is the default if no option is specified.

### **-eot [count]**

Lists all of the processes that are scheduled to execute at the end of the current simulation time.

The -eot option shows the processes that are scheduled to execute after normal activity for the current time has stabilized. These processes include postponed processes, non-blocking assignments, and \$monitor statements. Postponed processes and \$monitor statements are guaranteed to be the last processes run in the current time, but other end-of-time process activity may cause further delta cycles.

Include a *count* argument to limit the number of processes that are displayed. For example:

```
ncsim> process -eot 5
```

### **-next [count]**

Lists all of the processes that are scheduled to execute at the current simulation time.

Include a *count* argument to limit the number of processes that are displayed. For example:

```
ncsim> process -next 2
```

## process Command Examples

The Verilog source code used in the examples in this section is shown below. This source code was compiled with the `-linedebug` option.

```
module top_module;
reg[1:0]x, y, z;
wire [1:0]sum, co;
initial
    $monitor($time,, "x=%b y=%b z=%b sum=%b co=%b", x, y, z, sum, co);
initial
begin
    x=2'b00;
    y=2'b00;
    z=2'b00;
end
always
    x=#10 x+1;
always
    y=#40 y+1;
always
    z=#160 z+1;
endmodule
```

The `process` command with no command-line option is the same as `process -current`. This command displays the full pathname of the executing process, as well as the file name and the source code line number.

```
% ncsim worklib.top_module
ncsim> process
Executing Process:
[./test.v, 6, top_module]      initial
```

In the following command, the `-all` option shows the currently executing process and all scheduled processes, including those that are scheduled at the end of the current simulation time.

```
ncsim> run -process
./test.v:11      x=2'b00;
ncsim> process -all
Executing Process:
[./test.v, 9, top_module]      initial
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
Scheduled Process(es):
[./test.v, 16, top_module]      always
[./test.v, 19, top_module]      always
[./test.v, 22, top_module]      always

End-of-time Process(es):
[./test.v, 7, top_module]      $monitor($time,, "x=%b y=%b z=%b sum=%b          co=%b",
x, y, z, sum, co)
```

The following command line includes a *count* argument to limit the number of processes that are displayed.

```
ncsim> process -all 1
Executing Process:
[./test.v, 9, top_module]      initial
```

```
Scheduled Process(es):
[./test.v, 16, top_module]      always

End-of-time Process(es):
[./test.v, 7, top_module]      $monitor($time,, "x=%b y=%b z=%b sum=%b co=%b", x, y,
z, sum, co)
```

The following command shows the next two processes that are scheduled to execute.

```
ncsim> process -next 2
Scheduled Process(es):
[./test.v, 16, top_module]      always
[./test.v, 19, top_module]      always
```

The following command shows the processes that are scheduled to execute at the end of the current simulation time.

```
ncsim> process -eot
End-of-time Process(es):
[./test.v, 7, top_module]      $monitor($time,, "x=%b y=%b z=%b sum=%b co=%b", x, y,
z, sum, co)
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following example illustrates the output of the `process -all` command for a mixed-language design.

```
ncsim> process -all
ncsim> process -all
Executing Process:
[File: tb.vhd, Line: 153]      :$PROCESS_001

Scheduled Process(es):
[File: tb.vhd, Line: 152]      :$PROCESS_000
[File: tb.vhd, Line: 134]      :gen_quarters
[File: tb.vhd, Line: 122]      :gen_dimes
[File: tb.vhd, Line: 114]      :gen_nickels
...
...
...
[File: DRINK.vhd, Line: 356]    :top:$PROCESS_002
[File: DRINK.vhd, Line: 355]    :top:$PROCESS_001
[File: DRINK.vhd, Line: 354]    :top:$PROCESS_000
module DRINK_MACHINE:

[File: ./SOURCES/drink_machine.v, Line: 54]      initial $sdf_annotation( "SDFFILE" )

End-of-time Process(es):
No Processes Scheduled
```

## release

The `release` command releases any force set on the specified object(s). Releasing a force causes the value to immediately return to the value that would have been there if the force hadn't been blocking transactions. Use the `-keepvalue` option if you want to release the forced object, but retain the forced value.

This command will release any force, whether it was created by a `force` command or by a Verilog `force` procedural statement during simulation. The behavior is the same as that of a Verilog `release` statement.

Objects specified as arguments to the `release` command must have write access. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details on specifying access to simulation objects.

The following objects cannot be forced to a value with the `force` command and, therefore, cannot be specified as the object in a `release` command.

- memory
- memory element
- bit-select or part-select of a register
- bit-select or part-select of a unexpanded wire
- VHDL variable

See “[Forcing and Releasing Signal Values](#)” on page 435 for more information.

### release Command Syntax

```
release [-keepvalue] object_name ...
```

You can use wildcard characters in the argument to a `release` command.

- The asterisk ( \* ) matches any number of characters.
- The question mark ( ? ) matches any one character.

You cannot use wildcard characters inside escaped names.

See “[Using Wildcards Characters in Tcl Commands](#)” on page 495 for more information on using wildcards.

## release Command Modifiers and Options

### **-keepvalue**

Release the forced object, but retain the forced value.

## release Command Examples

The following command removes a force set on object `x`.

```
ncsim> release x
```

The following command removes a force set on object `:top:DISPENSE_tempsig`.

```
ncsim> release :top:DISPENSE_tempsig
```

The following command releases two objects: `w[ 0 ]` and `r`.

```
ncsim> release w[ 0 ] r
```

The following command releases all forces applied on objects in the current scope that have names that end in `td`.

```
ncsim> release *td
```

## **reset**

The `reset` command resets the currently loaded model to its original state at time zero. The time-zero snapshot, created by the elaborator, must still be available.

The Tcl debug environment remains the same as much as possible after a reset.

- Tcl variables remain as they were before the reset.
- SHM and VCD databases remain open, and probes remain set.  
**Note:** VCD databases created with the `$dumpvars` call in Verilog source code are closed when you reset.
- Breakpoints remain set.
- Watch Windows and the SimVision Waveform Viewer window remain the same.

Forces and deposits in effect at the time you issue the `reset` command are removed.

See “[Saving, Restarting, Resetting, and Reinvoking a Simulation](#)” on page 327 for more information.

### **reset Command Syntax**

```
reset
```

### **reset Command Modifiers and Options**

None.

### **reset Command Examples**

The following command resets the currently loaded model to its original state at time zero.

```
ncsim> reset
```

## restart

The `restart` command replaces the currently simulating snapshot with another snapshot of the same elaborated design.

You must specify a snapshot name with the `restart` command, and the specified snapshot must be a snapshot created by the `save` command. (See “[save](#)” on page 602 for details on this command.)

The snapshot name is interpreted the same way as the snapshot name on the *ncsim* command line, with the addition that you can give only the view name preceded by a colon if you want to load a snapshot that is a view of the currently loaded cell. For example:

`restart top`

Restarts [*lib.*]top[:*view*]

If the view name is omitted, there must be only one snapshot of the given cell, otherwise the snapshot name is ambiguous. In this case, an error message is issued, and a list of available snapshots is printed.

`restart top:ckpt`

Restarts [*lib.*]top:ckpt

`restart :ckpt`

Restarts [*lib.*][*cell*]:ckpt

An error message is issued if the snapshot specified on the command line is not a snapshot of the design hierarchy that is currently loaded. That is, you cannot use the `restart` command to load a snapshot of a different elaborated design or one that comes from a different elaborated design. To load a different model, exit *ncsim* and then invoke it with the new snapshot.

When you restart with a saved snapshot in the same simulation session:

- SHM databases remain open and all probes remain set.
- Breakpoints set at the time that you execute the `restart` remain set.

**Note:** If you set a breakpoint that triggers, for example, every 10 ns (that is, at time 10, 20, 30, and so on) and restart with a snapshot saved at time 15, the breakpoint triggers at 20, 30, and so on, not at time 25, 35, and so on.

- Forces and deposits in effect at the time you issue a `save` command are still in effect when you restart.

If you exit the simulation and then invoke the simulator with a saved snapshot, databases are closed. Any probes and breakpoints are deleted. If you want to restore the full Tcl debug

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

environment when you restart, make sure that you save the environment with the `save -environment` command. This command creates a Tcl script that captures the current breakpoints, databases, probes, aliases, and predefined Tcl variable values. You can then use the `Tcl source` command after restarting or the `-input` option when you invoke the simulator to execute the script. For example,

```
% ncsim top
ncsim> (Open a database, set probes, set breakpoints, deposits,
        forces, etc.)
ncsim> run 100 ns
ncsim> save worklib.top:ckpt1
ncsim> save -environment ckpt1.tcl
ncsim> exit
% ncsim -tcl worklib.top:ckpt1
ncsim> source ckpt1.tcl
```

## restart Command Syntax

```
restart snapshot_name
restart -show
```

## restart Command Modifiers and Options

### **-show**

List the names of all snapshots that can currently be used as the argument to the `restart` command.

## restart Command Examples

In the following example, a `save` command is issued to save the simulation state as a view of the currently loaded cell, `top`. This snapshot can be loaded using either of the following two `restart` commands.

```
ncsim> save top:ckpt1
ncsim> restart top:ckpt1
ncsim> restart :ckpt1
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

In the following example, a `save` command is issued to save the simulation state as a view of the currently loaded cell, `top`. A second `save` command is issued to save the Tcl debug environment. If you exit the simulator, you can restart with the saved snapshot and then restore the debug settings by sourcing the script created with the `save -environment` command.

```
ncsim> save top:ckpt1
ncsim> save -environment top_ckpt1.env
ncsim> exit
% ncsim -tcl :ckpt1
ncsim> source top_ckpt1.env
```

The following command reloads the snapshot of the given cell, `top`. Because the view name is not specified, the snapshot name is ambiguous if there is more than one view, and an error message is issued.

```
ncsim> restart top
```

The following `restart` command results in an error because you are trying to replace the currently simulating snapshot (`asic1:ckpt1`) with another snapshot of a different elaborated design (`asic2:ckpt1`). You can only restart snapshots of the same elaborated design.

```
% ncsim -tcl asic1:ckpt1
ncsim> restart asic2:ckpt1
```

The following command lists all of the snapshots you can currently load with the `restart` command.

```
ncsim> restart -show
otherlib.board:module
worklib.board:ckpt1
worklib.board:ckpt2
```

## run

The `run` command starts simulation or resumes a previously halted simulation. You can:

- Run until an interrupt, such as a breakpoint or error, occurs or until simulation completes (the `run` command with no modifiers or arguments).
- Run one behavioral statement, stepping over subprogram calls (`-next`).
- Run one behavioral statement, stepping into subprogram calls (`-step`).
- Run until the current subprogram ends (`-return`).
- Run to a specified timepoint or for a specified length of time (`-timepoint`).
- Run to the beginning of the next delta cycle or to a specified delta cycle (`-delta`).
- Run to the beginning of the next phase of the simulation cycle (`-phase`).
- Run until the beginning of the next scheduled process or to the beginning of the next delta cycle, whichever comes first (`-process`).

See “[Starting a Simulation](#)” on page 325 for more information.

## run Command Syntax

```
run
  -clean
  -delta [cycle_spec]
  -next
  -phase
  -process
  -return
  -step
  [-timepoint] [time_spec] [-absolute | -relative]
```

## run Command Modifiers and Options

### **-clean**

Run the simulation to the next point at which it is possible to create a checkpoint snapshot with the `save -simulation` command. (See [“save” on page 602](#) for details.)

### **-delta [cycle\_spec]**

Run the simulation for the specified number of delta cycles. If no *cycle\_spec* argument is specified, run the simulation to the beginning of the next delta cycle. A `run -delta` command is the same as `run -delta 1`.

### **-next**

Run one behavioral statement, stepping over any subprogram calls.

In some cases, a `run -next` command produces results that are similar to a `run -step` command when the current execution point is a VHDL non-zero WAIT statement. For example, if the current execution point is a statement such as `wait for 10 ns;`, another process may be scheduled to run at the current simulation time while the current process is suspended because of the WAIT statement. A `run -next` command executes the next behavioral statement, and the simulation stops in the scheduled process. If you want to run to the next executable line in the source code after the WAIT statement, you can set a line breakpoint on that line and enter a `run` command.

### **-phase**

Run to the beginning of the next phase of the simulation cycle. A simulation cycle consists of two phases: signal evaluation and process execution.

### **-process**

Run until the beginning of the next scheduled process or to the beginning of the next delta cycle, whichever comes first.

In VHDL, a process is a process statement. In Verilog it is an `always` block, an `initial` block, or some other behavior that can be scheduled to run.

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

#### **-return**

Run until the current subprogram (task, function, procedure) returns.

#### **-step**

Run one behavioral statement, stepping into subprogram calls.

#### **[*-timepoint*] [*time\_spec*] [*-absolute* | *-relative*]**

Run until the specified time is reached. The time specification can be absolute or relative. Relative is the default.

In addition to time units such as fs, ps, ns, us, and so on, you can use `deltas` as the unit. For example,

```
ncsim> run 10 deltas
```

This is the same as `run -delta 10`.

If you include a time specification and a breakpoint or interrupt stops simulation before the specified time is reached, the time specification is thrown away. For example, in the following sequence of commands, the last `run` command will not stop the simulation at 500 ns.

```
ncsim> stop -object x
Created stop 1
ncsim> run 500 ns
Stop 1 {x = 0} at 10 ns
ncsim> run
```

`run -timepoint` without a *time\_spec* argument runs the simulation until the next scheduled event.

## **run Command Examples**

The following command runs the simulation until an interrupt occurs or until simulation completes.

```
ncsim> run
```

The following command advances the simulation to 500 ns absolute time. The `-timepoint` option is not required.

```
ncsim> run -timepoint 500 ns -absolute
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following command advances the simulation 500 ns relative time. With a time specification, **-relative** is the default.

```
ncsim> run 500 ns
```

The following command runs one behavioral statement, stepping into any subprogram calls.

```
ncsim> run -step
```

The following command runs one behavioral statement, stepping over any subprogram calls.

```
ncsim> run -next
```

The following command runs until the current subprogram returns. The subprogram can be a task, function, or procedure.

```
ncsim> run -return
```

The following two commands are equivalent. They both run the simulation for 5 delta cycles.

```
ncsim> run -delta 5
```

```
ncsim> run 5 deltas
```

## save

The `save` command creates a snapshot of the current simulation state. You can then use the `restart` command to load the saved snapshot and resume simulation. (See “[restart](#)” on page 595 for details on the `restart` command.)

The current simulation state that is saved in the snapshot includes the simulation time and all object values, scheduled events, annotated delays, the contents of the memory allocated for access type values, and file pointers. It does not include aspects of the debugging environment such as breakpoints, probes, Tcl variables, and GUI configuration. PLI/VPI callbacks and handles are saved under certain circumstances. Please refer to the PLI/VPI manuals for details.

You cannot save a snapshot if the simulator is in the process of executing sequential HDL code. If the simulation is in a state that cannot be saved, you must use the `run -clean` command to run the simulation until the currently running sequential behavior (if any) suspends itself at a delay, event control, or a VHDL wait statement.

You must specify a snapshot name with the `save` command. The snapshot name can be specified using `[lib.]cell[:view]` notation, or, if you want the snapshot to be a new view of the currently loaded cell, you can specify just the view name preceded by a colon. For example, if you are simulating `worklib.top:rtl`,

<code>save ckpt1</code>	Saves <code>worklib.ckpt1:rtl</code>
<code>save top:ckpt1</code>	Saves <code>worklib.top:ckpt1</code>
<code>save otherlib.top</code>	Saves <code>otherlib.top:rtl</code>
<code>save :ckpt1</code>	Saves <code>worklib.top:ckpt1</code>

The snapshot name must be a simple name containing only letters, numbers and underscores.

**Note:** The Windows NT and Linux Red Hat (6.1 and 6.2) operating systems impose a two gigabyte limit on the size of a file. If a library database exceeds this limit, you will not be able to add objects to the database. If you save many snapshot checkpoints to unique views in a single library, this file size limit could be exceeded. If you reach this limit, you can:

- Use `save -overwrite` to overwrite an existing snapshot. For example,  
`ncsim> save -simulation -overwrite snap1`
- Save snapshots to a separate library. For example,

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
% mkdir INCA_libs/snaplib  
% ncsim -f ncsim.args  
ncsim> run 1000 ns  
ncsim> save -simulation snaplib.snap1  
ncsim> run 1000 ns  
ncsim> save -simulation snaplib.snap2
```

- Remove snapshots using the *ncrm* utility. For example,

```
% ncrm -snapshot worklib.snap1
```

The state of the Tcl debug environment is not part of the simulation that is saved in a snapshot. To save the debug environment, you must issue a separate `save -environment` command. This command creates a Tcl script that captures the current breakpoints, databases, probes, aliases, and predefined Tcl variable values. You can then restore the environment by executing this script with the `Tcl source` command, or you can use the `-input` option when you invoke the simulator.

The `save -commands` command is the same as `save -environment`.

For example:

```
ncsim> save :ckpt1  
ncsim> save -environment ckpt1.tcl  
ncsim> restart :ckpt1  
ncsim> source ckpt1.tcl  
(or: % ncsim -tcl cell:ckpt1 -input ckpt1.tcl)
```

**Note:** These scripts are meant to be sourced into an empty environment (that is, an environment with no breakpoints, no probes, no databases). If you invoke the simulator, set some breakpoints and probes, and then source a script that contains commands to set breakpoints and probes, the simulator will probably generate errors telling you that some commands in the script could not be executed. These errors are due to name conflicts. For example, you may have set a breakpoint that received the default name “1”, and the command in the script is trying to create a breakpoint with the same name. You can, of course, give your breakpoints unique names to avoid this problem. You can also edit the scripts to make them work the way you would like them to work.

See “[Saving, Restarting, Resetting, and Reinvoking a Simulation](#)” on page 327 for more information.

## **save Command Syntax**

```
save [-simulation] snapshot_name [-overwrite]
```

```
save -environment [filename]
```

```
save -commands [filename]
```

## **save Command Modifiers and Options**

### **-commands [filename]**

save -commands is the same as save -environment.

### **-environment [filename]**

Create a Tcl script that captures the current breakpoints, databases, probes, aliases, and predefined Tcl variable values. The *filename* argument is optional. If no file name is specified, the script is written to standard output.

### **[-simulation] snapshot\_name**

Create a snapshot of the current simulation state. This option is the default.

### **-overwrite**

Overwrite an existing snapshot.

## **save Command Examples**

The following command saves the simulation state in *lib.cell1:ckpt1*, where *lib* is the name of the current work library, and *cell1* is the cell name of the currently loaded snapshot.

```
ncsim> save -simulation :ckpt1
```

The following command saves the simulation state in *lib.top:ckpt1*.

```
ncsim> save top:ckpt1
```

The following command saves the simulation state in *lib.ckpt1:view\_name*, where *view\_name* is the view name that is currently being simulated.

```
ncsim> save ckpt1
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following example illustrates how to use the `save`, `restart`, and `reset` commands.

```
% ncsim -input run.vc harddrive
ncsim: v2.1.(p1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
Loading snapshot worklib.hardrive:module ..... Done

ncsim> database -open waves -default -shm      ;# Open default SHM
                                ;# dbase called waves.
Created default SHM database waves
ncsim> probe -create -all -database waves      ;# Probe all signals
                                ;# in current scope.
Created probe 1
ncsim> stop -create -object clr      ;# Create an object breakpoint and a time
                                ;# breakpoint at absolute time 200.
Created stop 1
ncsim> stop -create -time -absolute 200 ns
Created stop 2
ncsim> run
0 FS + 0 (stop 1: harddrive.clr = 1)
./harddrive.v:12    clr = 1;
ncsim> run
at time 50 clr =1 data= 0 q= x
at time 150 clr =1 data= 1 q= 0
200 NS + 0 (stop 2)
ncsim> save :ckpt1          ;# Save the simulation state at 200 ns.
Saved snapshot worklib.hardrive:ckpt1
ncsim> stop -create -time -relative 300 ns      ;# Create a breakpoint to stop
                                ;# every 300 ns.

Created stop 3
ncsim> run
at time 250 clr =1 data= 2 q= 1
at time 350 clr =1 data= 3 q= 2
at time 450 clr =1 data= 4 q= 3
500 NS + 0 (stop 3)
ncsim> save :ckpt2          ;# Save another snapshot at time 500 ns.
Saved snapshot worklib.hardrive:ckpt2
ncsim> run
at time 550 clr =1 data= 5 q= 4
at time 650 clr =1 data= 6 q= 5
at time 750 clr =1 data= 7 q= 6
800 NS + 0 (stop 3)
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
ncsim> restart :ckpt1          ;# Use the restart command to load
;# worklib.hardrive.ckpt1.

Loaded snapshot worklib.hardrive:ckpt1
ncsim> time
200 NS           ;# Simulation is at 200 ns. Two breakpoints are still set
;# (the breakpoint set for absolute time 200 was deleted
;# automatically when it triggered). The SHM database has
;# been closed and all probes deleted.

ncsim> stop -show
1      Enabled Object harddrive.clr
3      Enabled Time 500 NS (every 300 NS)
ncsim> database -show
No databases are open
ncsim> run
at time 250 clr =1 data= 2 q= 1
at time 350 clr =1 data= 3 q= 2
at time 450 clr =1 data= 4 q= 3
500 NS + 0 (stop 3)
ncsim> reset          ;# Use the reset command to reset the model
;# to its original state at time zero.
;# Notice that both breakpoints are still set.

Loaded snapshot worklib.hardrive:module
ncsim> time
0 FS
ncsim> stop -show
1      Enabled      Object harddrive.clr
3      Enabled      Time 200 NS (every 300 NS)
ncsim>
```

The following example illustrates how to use the `save -environment` command.

```
% ncsim -tcl harddrive
ncsim: v2.1.(p1): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
Loading snapshot worklib.hardrive:module ..... Done
;# Set a line breakpoint, an object breakpoint, and create a probe.
;# The probe command creates a default SHM database.

ncsim> stop -create -line 32
Created stop 1
ncsim> stop -create -object harddrive.clk
Created stop 2
ncsim> probe -create -shm harddrive.data
Created default SHM database ncsim.shm
Created probe 1
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
ncsim> run
0 FS + 0 (stop 2: harddrive.clk = 0)
./harddrive.v:13      clk = 0;
ncsim> run
50 NS + 0 (stop 2: harddrive.clk = 1)
./harddrive.v:16 always #50 clk = ~clk;
ncsim> save -environment env1.env          ;# Save debug settings in a file
                                                ;# called env1.env.
ncsim> more env1.env                      ;# The file env1.env contains
                                                ;# commands to recreate debug settings.
set assert_report_level {note}
set assert_stop_level {error}
set autoscope {yes}
set display_unit {auto}
set tcl_prompt1 {puts -nonewline "ncsim> "}
set tcl_prompt2 {puts -nonewline "> "}
set time_unit {module}
set vlog_format {%h}
set assert_1164_warnings {yes}
stop -create -name 1 -line 32 harddrive
stop -create -name 2 -object harddrive.clk
database -open -shm -into ncsim.shm ncsim.shm -default
probe -create -name 1 -database ncsim.shm harddrive.data
scope -set harddrive
ncsim> exit                                ;# Exit and then reinvoke the simulator.
% ncsim -tcl harddrive
ncsim: v1.2.(b9): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
Loading snapshot worklib.harddrive:module ..... Done
ncsim> stop -show
No stops set
ncsim> source env1.env                      ;# Source the script env1.env.
ncsim>
;# Show the status of breakpoints, probes, and databases.
ncsim> stop -show
1      Enabled      Line: ./harddrive.v:32 (scope: harddrive)
2      Enabled      Object harddrive.clk
ncsim> probe -show
1      Enabled      harddrive.data (database: ncsim.shm) -shm
ncsim> database -show
ncsim.shm      Enabled      (file: ncsim.shm) (SHM) (default)
```

## scope

The `scope` command lets you:

- Set the current debug scope (`-set`)
- Describe items declared within a scope (`-describe`)
- Display the drivers of objects declared within a scope (`-drivers`)
- Print the source code, or part of the source code, for a scope (`-list`)
- Display scope information (`-show`)

See “[Traversing the Model Hierarchy](#)” on page 424 for more information on setting the debug scope.

### scope Command Syntax

```
scope [-set] [scope_name]
      -up

      -describe [scope_name]
      -names
      -sort {name | kind | declaration}

      -drivers [scope_name]

      -list [line | start_line end_line] [scope_name]

      -show
```

## scope Command Modifiers and Options

### **-describe [scope\_name]**

Describes all objects declared within the specified scope. If no scope is specified, objects in the current debug scope are described.

For objects without read access, the output of `scope -describe` does not include the object's value. For objects that have read access but no write access, the string (-W) is included in the output. For objects with neither read nor write access, the string (-RW) is included in the output. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details on specifying access to simulation objects.

### **-names**

Displays only the names of each declared item in the scope. This option is used with the `-describe` modifier.

### **-sort {name | kind | declaration}**

Specifies the sort order. This option is used with the `-describe` modifier.

There are three possible arguments to the `-sort` option:

- name—Sort alphabetically by name.
- kind—Sort by declaration type (reg, wire, instance, etc.).
- declaration—Sort by the order in which objects are declared in the source code.

### **-drivers [scope\_name]**

Shows the drivers of each object declared within the specified scope. If no scope is specified, the drivers of objects in the current debug scope are displayed.

The output of `scope -drivers` includes only the objects that have read access. However, even if an object has read access, its drivers may have been collapsed, combined, or optimized away, and the output of the command might indicate that the object has no drivers. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details on specifying access to simulation objects.

### **-list [*line* | *start\_line end\_line*] [*scope\_name*]**

Prints lines of source code for the specified scope, or for the current debug scope if no scope is specified.

You can follow the `-list` modifier with:

- No range of lines to print all lines for the scope.
- One line number to display that line of source text.
- Two line numbers to display the text between those two line numbers. You can use a dash ( - ) for either the `start_line` or the `end_line`.

### **-set [*scope\_name*]**

Sets the current debug scope to the specified scope. If no scope or other option is given, the name of the current scope is printed.

The `-set` modifier is optional.

### **-up**

Sets the debug scope to one level up the hierarchy from the current scope.

### **-show**

Shows scope information, including the current debug scope, instances within the debug scope, and top-level modules in the currently loaded model.

If the current debug scope is a subprogram on a VHDL call stack, the output of the `scope -show` command or of the `scope` command (with no options or arguments) shows the current scope as `:process_scope[nest_level]`. For example, if the current scope is a subprogram called `function1`, which is at nest-level 1, the output of these commands shows the current scope as:

```
:process1[1]
```

In the Scope Region on the SimControl window, however, the scope is shown using a different string. For example, the scope shown above would be shown on the SimControl window as one of the following, depending on the location of the function definition:

```
:process1:function1  
:function1
```

## scope Command Examples

The following example prints the name of the current scope.

```
ncsim> scope
```

The following example sets the debug scope to scope u1. The `-set` modifier is not required.

```
ncsim> scope -set u1
```

The following example moves the debug scope up one level in the hierarchy.

```
ncsim> scope -up
```

In the following VHDL example, the `stack -show` command displays the current call stack. The process (process1) is displayed as nest-level 0, the base of the stack. The subprogram function1 is :process1[1], and the subprogram function2 is :process1[2].

- The first `scope` command displays the current debug scope.
- The second `scope` command sets the debug scope to :process1.
- The third `scope` command sets the debug scope to the subprogram :process1[2] (that is, to function2).

```
ncsim> stop -subprogram function1
Created stop 1
ncsim> run
0 FS + 0 (stop 1: Subprogram :function1)
./test.vhd:36 tmp4_local := function2 (tmp4);
ncsim> run -step
./test.vhd:29 tmp5_local := tmp5 + 1;
ncsim> stack -show          # Display the current call stack
2: Scope: :process1[2] Subprogram:@work.e(a):function2
   File: /usr1/belanger/inca/vhdl/subprogram_debug	scope_test/test.vhd
   Line: 29

1: Scope: :process1[1] Subprogram:@work.e(a):function1
   File: /usr1/belanger/inca/vhdl/subprogram_debug	scope_test/test.vhd
   Line: 36

0: Scope: :process1
   File: /usr1/belanger/inca/vhdl/subprogram_debug	scope_test/test.vhd
   Line: 52
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
ncsim> scope -show                      ;# Display the current debug scope
Directory of scopes at current scope level:

Current scope is (:process1[2])
Highest level modules:
    Top level VHDL design unit:
        entity (e:a)
    VHDL Package:
        STANDARD
        ATTRIBUTES
        std_logic_1164
        TEXTIO
ncsim> scope -set :process1              ;# Set scope to :process1
ncsim> scope -set :process1[2]           ;# Set scope to :process1[2]
                                         ;# i.e., function2
```

In the previous example, you can also use the `stack` command to set the debug scope to `:process1[2]` (that is, `function2`).

```
ncsim> scope -set :process1
ncsim> stack -set 2
```

The following command displays a list and a description of all objects declared in the current debug scope (a Verilog module).

```
ncsim> scope -describe
clr.....register = 1'hx
clk.....register = 1'hx
data.....register [3:0] = 4'hx
q.....wire [3:0] (wire/tri) = 4'hx
end_first_pass...named event
h1.....instance of module hardreg
```

The following command displays a list and a description of all objects declared in the current debug scope (a VHDL architecture).

```
ncsim> scope -describe
top.....component instantiation
load_nickels.....process statement
load_dimes.....process statement
load_cans.....process statement
load_action.....process statement
gen_clk.....process statement
gen_reset.....process statement
gen_nickels.....process statement
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
gen_dimes.....process statement
gen_quarters.....process statement
$PROCESS_000.....process statement
$PROCESS_001.....process statement
stoppit.....signal : BOOLEAN = TRUE
t_NICKEL_OUT.....signal : std_logic = '0'
t_EMPTY.....signal : std_logic = '1'
t_EXACT_CHANGE...signal : std_logic = '0'
t_TWO_DIME_OUT...signal : std_logic = 'Z'
...
...
t_NICKELS.....signal : std_logic_vector(7 downto 0) = "11111111"
t_RESET.....signal : std_logic = '0'
```

The following command lists the names of all objects declared in the current debug scope. No description is included.

```
ncsim> scope -describe -names
clr clk data q end_first_pass h1
```

The following example displays a list and a description of all objects declared in the current debug scope. Objects are listed in alphabetical order.

```
ncsim> scope -describe -sort name
```

The following command displays a list and a description of all objects declared in the current debug scope. Objects are sorted by type of declaration.

```
ncsim> scope -describe -sort kind
```

The following example displays a list and a description of all objects declared in scope h1. Objects are listed in the order in which they were declared in the source code.

```
ncsim> scope -describe -sort declaration h1
clk.....input (wire/tri) = StX
clrb.....input (wire/tri) = StX
d.....input [3:0] (wire/tri) = 4'hx
q.....output [3:0] (wire/tri) = 4'hx
f1.....instance of module flop
f2.....instance of module flop
f3.....instance of module flop
f4.....instance of module flop
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following command shows the drivers for all objects declared in scope h1.

```
ncsim> scope -drivers h1
clk.....input (wire/tri) = St1
    St1 <- (harddrive.h1) input port 2, bit 0 (./harddrive.v:8)
clrb.....input (wire/tri) = St1
    St1 <- (harddrive.h1) input port 3, bit 0 (./harddrive.v:8)
d.....input [3:0] (wire/tri) = 4'h2
[3] = St0
    St0 <- (harddrive.h1) input port 1, bit 3 (./harddrive.v:8)
[2] = St0
    St0 <- (harddrive.h1) input port 1, bit 2 (./harddrive.v:8)
[1] = St1
    St1 <- (harddrive.h1) input port 1, bit 1 (./harddrive.v:8)
[0] = St0
    St0 <- (harddrive.h1) input port 1, bit 0 (./harddrive.v:8)
q.....output [3:0] (wire/tri) = 4'h1
[3] = St0
    St0 <- (harddrive.h1.f4) nd7 (q, e, qb)
[2] = St0
    St0 <- (harddrive.h1.f3) nd7 (q, e, qb)
[1] = St0
    St0 <- (harddrive.h1.f2) nd7 (q, e, qb)
[0] = St1
    St1 <- (harddrive.h1.f1) nd7 (q, e, qb)
```

In the following example, the design was elaborated using the default access level (no read or write access to simulation objects). Notice the difference in output between this example and the previous example, where the design was elaborated with full access (ncelab -access +r+w). In this example, only the drivers for wires and registers with read access are shown.

```
ncsim> scope -drivers h1
q.....output [3:0]
q[3] (wire/tri) = St0
    St0 <- (harddrive.h1.f4) nd7 (q, e, qb)
q[2] (wire/tri) = St0
    St0 <- (harddrive.h1.f3) nd7 (q, e, qb)
q[1] (wire/tri) = St0
    St0 <- (harddrive.h1.f2) nd7 (q, e, qb)
q[0] (wire/tri) = St1
    St1 <- (harddrive.h1.f1) nd7 (q, e, qb)
```

The following example lists the source for the current debug scope.

```
ncsim> scope -list
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following example lists the source for scope u1.

```
ncsim> scope -list u1
```

The following example displays line 12 of the source for the current debug scope.

```
ncsim> scope -list 12
```

The following example lists lines 10 through 15 of the source for the current debug scope.

```
ncsim> scope -list 10 15
```

The following command lists lines from the top of the module through line 10 of the source for the current debug scope.

```
ncsim> scope -list - 10
```

The following command lists lines of source for the current debug scope, beginning with line 30.

```
ncsim> scope -list 30 -
```

The following command shows the output of the `scope -describe` command when you run in regression mode (no read, write, or connectivity access to simulation objects) and some objects do not have read or write access.

```
ncsim> scope -describe h1
clk.....input (-RW)
clrb.....input (-RW)
d.....input [3:0]
  d[3] (-RW)
  d[2] (-RW)
  d[1] (-RW)
  d[0] (-RW)
q.....output [3:0]
  q[3] (wire/tri) = St0
  q[2] (wire/tri) = St0
  q[1] (wire/tri) = St0
  q[0] (wire/tri) = St1
f1.....instance of module flop
f2.....instance of module flop
f3.....instance of module flop
f4.....instance of module flop
```

## source

The `source` command lets you execute Tcl commands contained in a script file.

When *ncsim* has processed all the commands in the source file, or if you interrupt processing with CTRL/C, input reverts back to the terminal.

See “[Providing Interactive Commands from a File](#)” on page 335 for more information.

You can also execute Tcl commands in a script file by using the `input` command or by using the `-input` command-line option when you invoke the simulator. However, the behavior of the `source` command differs from the behavior of the `input` command or the `-input` option in the following ways:

- With the `source` command, execution of the commands in the script stops if a command generates an error. With the `input` command, the contents of the file are read in place of standard input at the next Tcl prompt, as if you had typed the commands at the command-line prompt. This means that errors do not stop the execution of commands in the script.
- The `input` command echoes commands to the screen as they are executed, along with any command output or error messages. The `source` command, on the other hand, displays the output of only the last command in the file. Output from the model (for example, the output of `$display`, `$monitor`, or `$strobe` tasks, or the output of stop points) is printed to the screen.

See “[input](#)” on page 560 for details on the `input` command. See [-input](#) for details on the `-input` option.

### source Command Syntax

```
source script_file
```

**Note:** You can specify only one *script\_file*.

### source Command Modifiers and Options

None.

## source Command Examples

The following example uses the command file named `set_break.inp` to show how the `source` command displays output from only the last command in the file. The `set_break.inp` file contains the following commands:

```
stop -create -line 27
run
value data
run 50
value data
run 50
value data
run
%> ncsim -tcl harddrive
ncsim: v1.0.(p2): (c) Copyright 1995, 1996 Cadence Design Systems, Inc.
ncsim>
ncsim> source set_break.inp
0 FS + 0 (stop 1) 27:      repeat (2)      ;# Output of the stop command
at time 50 clr =1 data= 0 q= x          ;# Output of $strobe task in model
at time 150 clr =1 data= 1 q= 0
at time 250 clr =1 data= 2 q= 1
...
at time 3150 clr =0 data=14 q= 0
at time 3250 clr =0 data=15 q= 0
at time 3350 clr =0 data=15 q= 0
Simulation complete via $finish(1) at time 3400 NS + 0 ;# Output of last command
                                                ;# (run)
ncsim>      ;# Control reverts to the terminal after the simulator
          ;# executes the last command.
```

## stack

The `stack` command helps you to debug VHDL descriptions that involve multiple subprogram calls. Using this command, you can:

- View the current call stack (`-show`).
- Set the current stack frame (`-set`) so that you can describe objects that are local to a subprogram, view object values, and set object values.

## stack Command Syntax

```
stack  
[-set] [{stack_level_spec | -down | -up}]  
  
[-show] [-levels stack_level_count]
```

## stack Command Modifiers and Options

### **-set [ { stack\_level\_spec | -down | -up } ]**

The `stack -set` command with no argument displays the call stack for the current debug scope. This is the same as `stack -show`.

If you specify an argument, the argument can be:

- `stack_level_spec`

Sets the current debug stack to the specified stack level. This applies only to the current debug scope, which must be a process scope.

The `stack_level_spec` argument specifies the call stack depth. This can be:

- An absolute value, which sets the context to the specified stack frame. For example:

```
stack -set 2
```

- A relative value, which moves the stack context up or down relative to the current stack frame. For example:

```
stack -set +2
```

stack -set -1

■ **-down**

Sets the debug stack one level down from the current level.

stack -set -down is the same as stack -set -1.

■ **-up**

Sets the debug stack one level up from the current level.

stack -set -up is the same as stack -set +1.

**-show [-levels *stack\_level\_count*]**

Displays the call stack for the current debug scope.

Include the -levels option to limit the depth of the call stack that is displayed.

## stack Command Examples

The following VHDL source code is used in the examples in this section:

```
LIBRARY IEEE;
USE IEEE.Std_logic_1164.all;
USE STD.TEXTIO.ALL;
entity e is
end;
architecture a of e is

procedure procedure1 (tmp1 : INOUT bit) is
begin
    tmp1 := not tmp1;
    return;
end procedure1;

procedure procedure2 (tmp2 : INOUT bit) is
begin
    if (tmp2 = '0')
    then
        tmp2 := not tmp2;
        return;
    end if;
end procedure2;
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
else
    procedure1(tmp2);
end if;
end procedure2;

function function2 (tmp5 : IN integer) return integer is
    variable tmp5_local : integer := tmp5;
begin
    tmp5_local := tmp5 + 1;
    return tmp5_local;
end function2;

function function1 (tmp4 : IN integer) return integer is
    variable tmp4_local : integer := tmp4;
begin
    tmp4_local := function2 (tmp4);
    return tmp4_local;
end function1;

begin
    process1 : process
        variable var1, var2, var3 : bit := '0';
        variable var4, var5, var6, var7, var8 : integer := 0;
begin
    var1 := '1';
    procedure1 (var1);
    var2 := '1';
    procedure2 (var2);
    procedure2 (var3);
    var5 := 1;
    var6 := function1 (var5);
    var7 := function2 (var6);
    wait;
end process;
end;
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

In the following example, the `stack -show` command displays the current call stack. The process (`process1`) is displayed as nest-level 0, the base of the stack.

After setting the scope to `process1`, a `stack -set` command sets the current debug scope to nest-level 1 (`function1`). The `stack -set -up` command sets the debug scope to nest-level 2 (`function2`).

```
ncsim> stop -subprogram function1
Created stop 1
ncsim> run
0 FS + 0 (stop 1: Subprogram :function1)
./test.vhd:36 tmp4_local := function2 (tmp4);
ncsim> run -step
./test.vhd:29 tmp5_local := tmp5 + 1;
ncsim>
ncsim> stack -show          ;# Display the current call stack
2: Scope: :process1[2] Subprogram:@work.e(a):function2
   File: /usr1/belanger/inca/vhdl/subprogram_debug/scope_test/test.vhd
   Line: 29

1: Scope: :process1[1] Subprogram:@work.e(a):function1
   File: /usr1/belanger/inca/vhdl/subprogram_debug/scope_test/test.vhd
   Line: 36

0: Scope: :process1
   File: /usr1/belanger/inca/vhdl/subprogram_debug/scope_test/test.vhd
   Line: 52
ncsim> stack -show -levels 2      ;# Display only two levels
2: Scope: :process1[2] Subprogram:@work.e(a):function2
   File: /usr1/belanger/inca/vhdl/subprogram_debug/scope_test/test.vhd
   Line: 29

1: Scope: :process1[1] Subprogram:@work.e(a):function1
   File: /usr1/belanger/inca/vhdl/subprogram_debug/scope_test/test.vhd
   Line: 36
ncsim>
ncsim> scope -show          ;# Display the current scope
Directory of scopes at current scope level:

Current scope is (:process1[2])
Highest level modules:
  Top level VHDL design unit:
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
entity (e:a)
VHDL Package:
  STANDARD
  ATTRIBUTES
  std_logic_1164
  TEXTIO

ncsim>
ncsim> scope -set :process1          ;# Set scope to process1
ncsim> stack -set 1                ;# Set scope to :process1[1]
                                (i.e., function1)
ncsim> scope -show
Directory of scopes at current scope level:

Current scope is (:process1[1])
...
...
...
ncsim>
ncsim> stack -set -up           ;# Set scope up one level, to :process1[2]
                                ;# This is the same as stack -set +1
ncsim> scope -show
Directory of scopes at current scope level:

Current scope is (:process1[2])
...
...
ncsim> value tmp5            ;# Display the value of :process1[2]:tmp5
1
ncsim> value :process1[1]:tmp4 ;# Display the value of :process1[1]:tmp4
1
```

## **status**

The **status** command displays memory and CPU usage statistics and shows the current simulation time.

### **status Command Syntax**

`status`

### **status Command Modifiers and Options**

None.

### **status Command Examples**

The following example shows the type of statistics displayed by the **status** command.

```
ncsim> status
Memory Usage - text: 3656824, static: 561600, dynamic: 835136, total: 5053560
CPU Usage - 12.21 seconds (user = 8)
Simulation Time - 19721 US + 0
```

## **stop**

The `stop` command creates or operates on a breakpoint. You can:

- Create various kinds of breakpoints (using the `-create` modifier followed by an option that specifies the breakpoint type)
- Display information on breakpoints (`-show`)
- Disable a breakpoint (`-disable`)
- Enable a previously disabled breakpoint (`-enable`)
- Delete a breakpoint (`-delete`)

See the following sections for more information:

- [“Setting a Condition Breakpoint”](#) on page 427
- [“Setting a Source Code Line Breakpoint”](#) on page 428
- [“Setting an Object Breakpoint”](#) on page 429
- [“Setting a Time Breakpoint”](#) on page 430
- [“Setting a Delta Breakpoint”](#) on page 431
- [“Setting a Process Breakpoint”](#) on page 431
- [“Setting a Subprogram Breakpoint”](#) on page 432
- [“Disabling, Enabling, Deleting, and Displaying Breakpoints”](#) on page 433

## stop Command Syntax

```
stop -create
      -condition {tcl_expression}

      -delta delta_cycle_number [-relative | -absolute]
          [-start delta_cycle_number]
          [-modulo delta_cycle_number]

      -line line_number
          { -unit unit_name | [scope_name] [-all] }
          [-file filename]

      -object object_names

      -process process_name

      -subprogram subprogram_name

      -time time_spec [-relative | -absolute]
          [-start time_spec]
          [-modulo time_spec]

          [-continue]
          [-delbreak count]
          [-execute command]
          [-if {tcl_expression}]
          [-name break_name]
          [-silent]
          [-skip count]

      -delete {break_name | pattern} ...
      -disable {break_name | pattern} ...
      -enable {break_name | pattern} ...
      -show [{break_name | pattern} ...]
```

The argument to `-delete`, `-disable`, `-enable`, or `-show` can be:

- a break name
- a list of break names
- a pattern
  - The asterisk ( `*` ) matches any number of characters
  - The question mark ( `?` ) matches any one character
  - `[characters]` matches any one of the characters
- Any combination of literal break names and patterns

## **stop Command Modifiers and Options**

The stop command has five modifiers:

- `-create`—Lets you create various types of breakpoints.
- `-delete`—Lets you delete a breakpoint.
- `-disable`—Lets you disable a breakpoint.
- `-enable`—Lets you enable a previously disabled breakpoint.
- `-show`—Lets you display information on breakpoints.

## **Creating a Breakpoint**

### **-create**

Creates a breakpoint. This modifier must be followed by an option that specifies the breakpoint type:

- `-condition`
- `-delta`
- `-line`
- `-object`
- `-process`
- `-subprogram`

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

#### ■ **-time**

#### **-condition {*tcl\_expression*}**

Sets a breakpoint that triggers when any object referenced in the *tcl\_expression* changes value (wires, signals, registers, and variables) or is written to (memories) AND the expression evaluates to true (non-zero, non-x, non-z).

The simulator does not support stop points on individual bits of registers. If a bit-select of a register appears in the expression, the simulator stops and evaluates the expression when any bit of that register changes value. The same holds true for compressed wires.

See “[Tcl Expressions as Arguments](#)” on page 641 for details on the format of conditional expressions.

Objects included in a -condition expression must have read access. An error is printed if the object does not have read access. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248.

#### **-continue**

Resumes the simulation after executing the breakpoint. The simulator will not go into interactive mode.

#### **-delbreak *count***

Deletes the breakpoint after it has triggered *count* number of times.

#### **-delta *delta\_cycle\_num* [-absolute] [-relative] [-start *delta\_cycle\_num*] [-modulo *delta\_cycle\_num*]**

Sets a breakpoint that triggers when the simulation delta cycle count reaches the specified delta cycle.

The delta cycle specification can be absolute or relative (the default). If absolute, the breakpoint is automatically deleted after the delta cycle is reached and the breakpoint triggers. If relative, the delta cycle specification is an interval, and the breakpoint stops the simulation every *n* delta cycles.

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

Use `-start` to specify the absolute delta cycle at which a repetitive breakpoint is to begin firing. If this cycle is before the current cycle, the first stop occurs at the next cycle at which it would have occurred had the stop been set at the cycle specified with `-start`.

The `-modulo` option is similar to `-start`. Use `-modulo` to specify the absolute delta cycle of the first stop cycle for a repeating delta cycle stop. This differs from `-start` only when the given cycle is more than one repeat interval in the future. In this case, the first stop occurs at a delta cycle less than or equal to one interval in the future such that a stop will eventually occur at the given cycle. For example, if you set a delta breakpoint to stop the simulation every 10 delta cycles, and specify `-modulo 15`, the simulation stops at delta cycle 5, 15, 25, and so on.

When you execute a `save -environment` command to save your debug environment, this option is written to the script to restore your delta breakpoint pattern.

See “[Setting a Delta Breakpoint](#)” on page 431 for more information on setting delta breakpoints.

#### **`-execute command`**

Executes the specified Tcl command when the breakpoint is triggered.

If the command that you want to execute requires an argument, enclose the command and its argument in curly braces.

You also can specify that you want to execute a list of commands. Separate the commands with a semi-colon. Tcl, however, displays only the output of the last command.

#### **`-if {tcl_expression}`**

Sets a condition on the breakpoint. The breakpoint will trigger only if the given Tcl boolean expression evaluates to true (non-zero, non-x, non-z). This option can be used with any breakpoint type. See “[Tcl Expressions as Arguments](#)” on page 641 for more information on the format of the `tcl_expression` argument.

Objects included in an `-if` expression must have read access. An error is printed if the object does not have read access. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details on specifying access to simulation objects.

**-line *line\_number* { -unit *unit\_name* | [*scope\_name*] [-all] } [-file *filename*]**

Sets a breakpoint that triggers when the specified line number is about to execute.

You must specify which design unit contains the line. There are two ways to do this:

- Use `-unit`. The stop will occur whenever the line number in the specified design unit is about to execute, no matter where in the design hierarchy that unit appears.
- Specify the name of a particular scope in the design hierarchy. This creates an instance-specific breakpoint. The breakpoint will occur only for that particular instance of the corresponding design unit, no matter where else it may appear in the design hierarchy. To create a breakpoint that is not instance-specific using the `scope_name` method, use the `-all` option. If the scope name is omitted, then the current debug scope is used.

The `-file` option specifies which of the source files that make up the specified design unit contains the specified line. This is necessary if the design unit has multiple source files.

**Note:** You must compile with the `-linedebug` option to enable the setting of line breakpoints.

See “[Setting a Source Code Line Breakpoint](#)” on page 428 for more information on line breakpoints.

**-name *break\_name***

Specifies a name for the breakpoint. This name can then be used to delete, disable, or enable the breakpoint. If you do not use `-name`, breakpoints are numbered sequentially.

**-object *object\_name***

Sets a breakpoint that triggers when the specified object changes value (wires, signals, registers, and variables) or is written to (memories).

You can use wildcard characters in the argument to a `stop -object` command.

- The asterisk ( \* ) matches any number of characters.
- The question mark ( ? ) matches any one character.

You cannot use wildcard characters inside escaped names.

See “[Using Wildcards Characters in Tcl Commands](#)” on page 495 for more information on using wildcards.

The object specified as the argument must have read access for the breakpoint to be created. An error is printed if the object does not have read access. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details on specifying access to simulation objects.

By default, all vectors are compressed, and a `stop -object` command can only be used on the entire object. You must elaborate the design with the `-expand` option (`ncelab -expand`) in order to set a breakpoint on a subelement of a vector Verilog wire or VHDL signal.

You cannot set a breakpoint on a VHDL subprogram object.

See “[Setting an Object Breakpoint](#)” on page 429 for more information on object breakpoints.

#### **`-process process_name`**

Sets a breakpoint that triggers when the specified VHDL named process starts executing or when it resumes executing after a wait statement.

**Note:** You must compile with the `-linedebug` option to enable the setting of process breakpoints.

See “[Setting a Process Breakpoint](#)” on page 431 for more information on process breakpoints.

#### **`-silent`**

Suppresses the display of the message that is printed when a breakpoint triggers.

#### **`-skip count`**

Tells the simulator to ignore the breakpoint for the first `count` times that it triggers.

You can use `-skip` to set a breakpoint on the  $n$ 'th occurrence of an event; in particular, you can use it to get inside `for` loops.

**-subprogram *subprogram\_name***

Sets a breakpoint that triggers when the specified subprogram is called.

**Note:** You must compile with the `-linedebug` option to enable the setting of subprogram breakpoints.

**-time *time\_spec* [-absolute] [-relative] [-start *time\_spec*] [-modulo *time\_spec*]**

Sets a breakpoint that triggers at the specified time. The time can be absolute or relative (the default). Absolute time breakpoints are automatically deleted after they trigger. Relative time breakpoints are periodic, stopping, for example, every 10 ns.

Use `-start` to specify the absolute simulation time at which a relative time breakpoint is to begin firing. If this time is before the current simulation time, the first stop occurs at the next future time at which it would have occurred had the stop been set at the time specified with `-start`.

The `-modulo` option is similar to `-start`. Use `-modulo` to specify the absolute simulation time of the first stop time for a repeating stop. This differs from `-start` only when the given time is more than one repeat interval in the future. In this case, the first stop occurs at a time less than or equal to one interval in the future such that a stop will eventually occur at the given time. For example, if you set a time breakpoint to stop the simulation every 100 ns, and specify `-modulo 250`, the simulation stops at time 50, 150, 250, and so on.

When you execute a `save -environment` command to save your debug environment, this option is written to the script to restore your time breakpoint pattern.

See “[Setting a Time Breakpoint](#)” on page 430 for more information.

## **Deleting a Breakpoint**

**-delete {*break\_name* | *pattern*} ...**

Deletes the breakpoint(s) specified by the argument. See “[Disabling, Enabling, Deleting, and Displaying Breakpoints](#)” on page 433 for more information.

## Disabling a Breakpoint

**-disable {*break\_name* | *pattern*} ...**

Disables the breakpoint(s) specified by the argument without deleting it. See “[Disabling, Enabling, Deleting, and Displaying Breakpoints](#)” on page 433 for more information.

## Enabling a Breakpoint

**-enable {*break\_name* | *pattern*} ...**

Enables the previously disabled breakpoint(s) specified by the argument. See “[Disabling, Enabling, Deleting, and Displaying Breakpoints](#)” on page 433 for more information.

## Displaying Information About Breakpoints

**-show [{*break\_name* | *pattern*} ...]**

Shows the status of the breakpoint(s) specified by the argument. If no breakpoint is specified, all breakpoints are shown. See “[Disabling, Enabling, Deleting, and Displaying Breakpoints](#)” on page 433 for more information.

## stop Command Examples

### Object Breakpoints

The following command creates a breakpoint that stops simulation when `sum` changes value. The `-create` modifier is not required. Because the `-name` option is not included to specify a breakpoint name, *ncsim* assigns a sequential number as the name. This breakpoint is called 1.

```
ncsim> stop -create -object sum  
Created stop 1
```

The following command creates breakpoints on all objects in the current scope.

```
ncsim> stop -object *  
Created stop 1
```

The following command creates breakpoints on all objects in the current scope that have names that start with `bus`.

```
ncsim> stop -object bus*
```

The following command creates breakpoints on all objects in the scope `top` that have two letters and that have names that start with the letter `c`.

```
ncsim> stop -object top.c?
```

The following command creates a breakpoint named `mybreak` that stops simulation when `sum` changes value.

```
ncsim> stop -object sum -name mybreak  
Created stop mybreak
```

The following command creates a breakpoint that triggers when `sum` changes value. The breakpoint is ignored the first 3 times it triggers.

```
ncsim> stop -object sum -skip 3
```

The following command creates a breakpoint that stops simulation when `clr` changes value. The `value data` command is executed when the breakpoint triggers. Because the `value` command requires an argument, it must be enclosed in curly braces.

```
ncsim> stop -object clr -execute {value data}
```

The following command creates a breakpoint that triggers when `clr` changes value. The `value data` command is executed when the breakpoint triggers. The `-continue` option prevents the simulator from entering interactive mode every time the stop triggers.

```
ncsim> stop -object clr -execute {value data} -continue
```

The following command creates an object breakpoint that triggers when `data` changes value. The `-delbreak` option specifies that the breakpoint is deleted after it triggers three times.

```
ncsim> stop -object data -continue -delbreak 3
```

The following command creates a breakpoint that triggers when `clk` changes value, but only if `clk` is high. See “[Tcl Expressions as Arguments](#)” on page 641 for details on the syntax of the argument to the `-if` option.

```
ncsim> stop -object clk -if {#clk == 1} -continue
```

The following command creates a breakpoint that triggers when `data[1]` has the value 1 and the time becomes greater than 3 ns.

```
stop -object data -if {#data[1] == 1 && [time ns -nounit] > 3}
```

The following command shows the error message that is displayed if you run in regression mode (no read, write, or connectivity access to simulation objects) and then try to set an object breakpoint on an object that does not have read access.

```
ncsim> stop -object clk
ncsim: *E,RDACRQ: Object does not have read access: harddrive.clk.
```

## Line Breakpoints

The following command creates a breakpoint that stops simulation when line number 10 in the current debug scope is about to execute.

```
ncsim> stop -line 10
```

The following command creates a breakpoint that stops simulation when line number 13 in scope `counter` is about to execute.

```
ncsim> stop -line 13 counter
```

In the following command, the `-all` option specifies that the stop is noninstance-specific. The breakpoint will occur on all scopes which are instances of the same module. For example if there are two instances of module `m16`, as follows, the breakpoint will trigger when line 13 in either `counter1` or `counter2` is about to execute.

```
module board;
<declarations>
m16 counter1 (...);
m16 counter2 (...);
<code>
endmodule
ncsim> stop -line 13 counter1 -all
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following command is equivalent to the command shown in the previous example. Both commands create non-instance-specific breakpoints.

```
ncsim> stop -line 13 -unit m16
```

In the following example, the `-file` option specifies which of the source files that make up the given scope (or the debug scope if none is given) contains the specified line. This is necessary if the scope has multiple source files.

```
ncsim> stop -line 13 counter -file foo.v
```

### Time Breakpoints

The following command creates a breakpoint that stops simulation at absolute time 200 ns. The breakpoint is automatically deleted after it triggers.

```
ncsim> stop -time 200 ns -absolute
```

The following command creates a repetitive breakpoint that stops the simulation every 200 ns and then executes the `value` command. The `-relative` option is the default for time breakpoints.

```
ncsim> stop -time 200 ns -relative -execute {value data}
```

The following command creates a repetitive breakpoint that stops the simulation every 200 ns. The `-start` option specifies the absolute time at which the breakpoint will start. For example, if the current simulation time is 300 ns, the breakpoint will stop the simulation at time 600, 800, 1000, and so on.

```
ncsim> stop -time 200 ns -start 600 ns
```

In the following example, assume that the current simulation time is 300 ns. The absolute time specified with `-start` is before the current simulation time. The first stop will occur at the next future time at which it would have occurred had the stop been set at the time specified with `-start`. In this example, the first stop will occur at time 450 ns.

```
ncsim> stop -time 200 ns -start 250 ns
```

The following example shows how the `-modulo` option is used to save a breakpoint pattern. Suppose that you simulate to time 300 ns and then set a repetitive breakpoint with the following command:

```
ncsim> stop -time 200 ns -start 350 ns
```

This command stops the simulation at time 350, 550, 750, and so on. If you then execute a `save -environment` command to save your debug environment, the following line is written to the script:

```
stop -create -name 1 -time 200 NS -relative -modulo 950 NS
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

If you then exit and re-enter the simulation and source the script containing this command, the breakpoint pattern is re-established. In this example, if you reinvoke the simulation and start at time 0, the breakpoint will trigger the first time at time 150. It will then trigger at 350, 550, 750, and so on.

The following command includes the `-if` option to set a breakpoint at time 100 ns (relative) if `data[1]` has the value 1.

```
ncsim> stop -time 100 ns -if {[#data[1]] == 1}
```

### Delta Breakpoints

The following command creates a breakpoint that stops the simulation when it reaches 20 delta cycles. The breakpoint is automatically deleted after it triggers.

```
ncsim> stop -delta 20 -absolute
```

The following command creates a repetitive breakpoint that stops the simulation every 10 delta cycles. The `-start` option specifies the absolute delta cycle at which the breakpoint will start. For example, if the current delta cycle count is 0, the breakpoint will stop the simulation when the delta cycle count is 30, 40, 50, and so on.

```
ncsim> stop -delta 10 -start 30
```

### Condition Breakpoints

In a condition breakpoint, the argument to the `-condition` option is a Tcl expression. See [“Tcl Expressions as Arguments”](#) on page 641 for more information on writing these expressions.

The following command sets a condition breakpoint that stops the simulation when `count`, the output of a 32-bit counter, has the value 100, decimal. The signal `count` is available from the top level of the hierarchy.

Verilog:

```
ncsim> stop -condition {[value %d top.count] = 100}
```

VHDL:

```
ncsim> stop -condition {[value %d :count] = 100}
```

If you are currently at the top level, you can omit the hierarchical path specification to `count`, and the two commands shown in the previous example could be written as follows:

```
ncsim> stop -condition {[value %d count] = 100}
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The `value` command uses the value of the `vlog_format` (or `vhdl_format`) variable. If you set the value of this variable to `%d`, the command shown in the previous example could be written as follows:

```
ncsim> stop -condition {[value count] = 100}
```

Instead of using the `value` command to get the value of `count` into the expression evaluator, you can use `#count`. Include the format specifier after the `#` sign.

```
ncsim> stop -condition {#%dcount = 100}
```

For Verilog, you can use the standard notation (for example `4'b0011`). For example, you can set the breakpoint on `count` as follows:

```
ncsim> stop -condition {#count = 32'd100}
ncsim> stop -condition {#count = 32'b00000000000000000000000000001100100}
```

VHDL does not have the same type of notation. Vectors must be enclosed in quotation marks, as shown in the next example.

```
ncsim> stop -condition {#count = "000000000000000000000000001100100"}
```

The following command sets a condition breakpoint that stops the simulation when bit 0 of `count` is 1. The expression is evaluated when any bit of `count` changes value. For VHDL, single-bit entities must be enclosed in single quotation marks.

Verilog:

```
ncsim> stop -condition {#count[0] == 1}
```

VHDL:

```
ncsim> stop -condition {#count(0) == '1'}
```

The following command is identical to the previous command. An explicit `value` command is used to get the value of `count`(bit 0) into the expression parser.

Verilog:

```
ncsim> stop -condition {[value %b count[0]] == 1'b1}
```

VHDL:

```
ncsim> stop -condition {[value %e count(0)] == '1'}
```

Note the use of `%e` in the VHDL example. The Tcl expression evaluator must know what enumeration type to use in the comparison expression, so at least one of the enumeration literals in the expression must be in the fully qualified format, which includes a path to the type declaration that defines the literal. The `value` command used with the `%e` format specifier returns the current value of a VHDL object in the fully qualified format. This value is substituted into the command line before the expression is passed to the Tcl expression evaluator.

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

In the following command, the `-if` option is used to conditionalize the condition breakpoint. This breakpoint will stop the simulation at the next positive edge of the clock if `en1` or `en2` is 1.

Verilog:

```
ncsim> stop -condition {#clock == 1} -if {#en1 || #en2}
```

VHDL:

```
ncsim> stop -condition {"#clk_n == '1'} -if {"enable=='1'|| #reset_n=='1'}
```

The following command stops the simulation at 5 ns (absolute time). After that, `clock` changes depending on the condition in the `if` expression, and this happens repeatedly every 5 ns. The `-continue` option is used to prevent the simulation from stopping every time the breakpoint triggers. VHDL requires use of the single quotation marks.

```
ncsim> stop -time 5 ns -start 5 ns -execute {if {#clk == '0'} {force clk '1'}  
else {force clk '0'}} -continue
```

The following command stops the simulation when the value of the specified signal has the value `x`. Notice that in the Tcl expression, the case-equality operator (`==`) is used. For this operator, bits that are unknown are included in the comparison, and the result of the expression is always 1 (true) or 0 (false).

```
ncsim> stop -create -condition {#top.load === 1'bx}
```

In the following command, the logical comparison operator (`=` or `==`) is used. These operators return the unknown value (`x`) if either operand is unknown. In a conditional expression, an unknown result is treated as false. Therefore, the following command will *not* stop the simulation when the signal has the value `x`.

```
ncsim> stop -create -condition {#top.load == 1'bx}
```

## Process Breakpoints

The following command sets a breakpoint that stops the simulation whenever the process called `:load_action` is executed.

```
ncsim> stop -process :load_action
```

## Subprogram Breakpoints

The following command sets a breakpoint that stops the simulation when the VHDL subprogram procedure1 is called.

```
ncsim> stop -subprogram procedure1
Created stop 1
ncsim> run
0 FS + 0 (stop 1: Subprogram :procedure1)
./scope_test.vhd:11           tmp1 := not tmp1;
```

The following command sets a breakpoint that stops the simulation when the VHDL subprogram function2 is called.

```
ncsim> stop -subprogram function2
Created stop 1
ncsim> run
0 FS + 0 (stop 1: Subprogram :function2)
./scope_test.vhd:29           tmp5_local := tmp5 + 1;
```

## Examples of Other stop Command Modifiers

The following command sequence illustrates the -show modifier. The first command creates a source line breakpoint called break1; the second creates an object breakpoint called break2. The third command shows the status of the two breakpoints.

```
ncsim> stop -line 12 -name break1
Created stop break1
ncsim> stop -object data -name break2
Created stop break2
ncsim> stop -show
break1 Enabled      Line: ./shortdrive.v:12 (scope: top)
break2 Enabled      Object top.data
```

In the following command sequence, breakpoint break1 is first disabled with the -disable modifier and then enabled with the -enable modifier.

```
ncsim> stop -show
break1 Enabled      Line: ./shortdrive.v:12 (scope: top)
break2 Enabled      Object top.data
ncsim> stop -disable break1
ncsim> stop -show
break1 Disabled     Line: ./shortdrive.v:12 (scope: top)
break2 Enabled      Object top.data
ncsim> stop -enable break1
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following command deletes breakpoint break1.

```
ncsim> stop -delete break1
```

To disable, enable, or delete the two breakpoints break1 and break2, any of the following commands could be used.

```
ncsim> stop -delete *1 *2  
ncsim> stop -delete break?  
ncsim> stop -delete br*
```

The following command displays information on any breakpoint beginning with v or b.

```
ncsim> stop -show {[vb]*}
```

## Tcl Expressions as Arguments

The `stop` command has two options that let you specify conditions. Both options require a Tcl expression argument.

- **-condition**

This option specifies that you are creating a condition breakpoint, as opposed to some other kind of breakpoint, such as a time or object breakpoint. A condition breakpoint triggers when any object named in the Tcl expression has an event that would trigger an object breakpoint and the expression evaluates to non-zero, non-x, or non-z.

- **-if**

This option can be used with any breakpoint type, including condition breakpoints. The Tcl expression argument is evaluated, and the stop triggers if the expression evaluates to non-zero, non-x, or non-z.

There are two general rules to keep in mind when writing the Tcl expression:

- Enclose the expression in braces to suppress immediate substitution of values.

*{tcl\_expression}*

**Note:** If you are using the SimVision analysis environment, these braces are included on the Set Break form.

In the following example, the value of `w[1]` would be substituted with its current value (`1'bx`, for example) if there were no braces. No object would be named in the expression by the time the `stop` command routine sees it, resulting in an error.

```
ncsim> stop -condition #w[1] == 1  
ncsim> stop -condition {#w[1] == 1}
```

- You must use either an explicit `value` command or the `#` character to get the object's value into the expression parser because the parser does not understand names. For example, the following command generates an error message.

```
ncsim> stop -time 100 ns -if {r[1] == 1}
```

Use the following commands:

**Verilog:**

```
ncsim> stop -time 100 ns -if {[value r[1]] == 1'b1}  
ncsim> stop -time 100 ns -if {[#r[1]] == 1}
```

**VHDL:**

```
ncsim> stop -time 100 ns -if {[value r(1)] == '1'}  
ncsim> stop -time 100 ns -if {[#r(1)] == '1'}
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

Format specifiers can be used with either the `value` command or the # sign. If you use the # sign, place the format specifier after the # sign. For example,

Verilog:

```
ncsim> stop -condition {[value %d out] = 12}
ncsim> stop -condition {#%dout = 12}
```

VHDL:

```
ncsim> stop -condition {[value %d out] = 12}
ncsim> stop -condition {#%dout = 1}
```

For VHDL, you must enclose vectors in quotation marks and single-bit entities in single quotation marks.

For example:

Verilog:

```
ncsim> stop -condition {#clock == 1}
```

VHDL:

```
ncsim> stop -condition {#clock == '1'}
```

Verilog:

```
ncsim> stop -condition {#count = 4'b0101}
```

VHDL:

```
ncsim> stop -condition {#clock = "0101"}
```

In a Tcl expression, the single-equality operator ( = ), which is used for assignment in Verilog, is a logical comparison operator. In a Tcl expression, = is the same as the Verilog logical comparison operator ( == ). The following two expressions are the same:

```
{ #top.load = 1'b1 }
{ #top.load == 1'b1 }
```

These operators return the unknown value (x) if either operand is unknown. In a conditional expression, an unknown result is treated as false. For example, in the following `stop` command, the expression returns an unknown result, and is, therefore, false. This conditional breakpoint will not trigger when the signal `top.load` has the value x.

```
ncsim> stop -create -condition {#top.load == 1'bx}
```

To set a breakpoint that will stop when the value is x, use the case equality operator ( === ). The result of the expression is always 1 (true) or 0 (false). For example:

```
ncsim> stop -create -condition {#top.load === 1'bx}
```

## **NC-Verilog Simulator Help**

### Using the Tcl Command-Line Interface

---

See [Appendix A, “Basics of Tcl,”](#) for more details on basic Tcl syntax and on the extensions to Tcl that have been added to handle types and operators of the Verilog and VHDL hardware description languages.

## strobe

The **strobe** command prints the values of specified VHDL or Verilog objects based on one of the following specifications:

- When a specified condition is true.
- When a specified signal changes value.
- At a specified time interval.

The **strobe** command is a Tcl procedure that uses the [stop](#) command to set a condition, object, or time breakpoint and then, when the breakpoint triggers, executes a [value](#) command to print out the values of the specified objects in tabular format.

By default, the simulator prints the values of the specified objects using the default format of the **value** command. You can set the **strobeFmt** variable to specify a global format. For example:

```
ncsim> set strobeFmt %b
```

**Note:** You must reset the strobe after setting the **strobeFmt** variable.

You can also specify a format for individual objects. Enclose the object-format pair in curly braces. For example:

```
ncsim> strobe -time 100 clk {data %b}
```

If you interrupt or stop the simulation (with **CTRL/C** or with an **assert** statement for example), and then continue the simulation, the simulator does not print the header in the tabular output. To display the header again, set the **strobeHeader** variable to 1, as follows:

```
ncsim> set strobeHeader 1
```

Simulation objects in the design must have read access. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details on specifying access to simulation objects.

You can only set one strobe at a time. Setting a new strobe automatically deletes the previous strobe.

## strobe Command Syntax

```
strobe {-time time_spec | -object object_spec | -condition condition_spec}
        object | {object format} [object | {object format} ...]

strobe -delete
strobe -help
```

## strobe Command Modifiers and Options

You must specify if you want to display signal values when a specified condition is true (-condition), when a specified signal changes value (-object), or at a specified time interval (-time).

**-condition condition\_spec object | {object format} [object |
{object format} ...]**

Display the values of the specified object(s) when the *condition\_spec* is true.

The *condition\_spec* argument is the same as in the [stop](#) command.

**-object object\_spec object | {object format} [object |
{object format} ...]**

Display the values of the specified object(s) when the object specified by the *object\_spec* argument changes value.

With -object, you can specify only one object to monitor for a change in value. For example, the following command displays the values of *y* and *z* when *x* changes value.

```
ncsim> strobe -object x y z
```

The following command displays the value of *y* in the default format and the value of *z* in binary when *x* changes value.

```
ncsim> strobe -object x y {z %b}
```

**-time time\_spec object | {object format} [object | {object format} ...]**

Display the values of the object(s) at the time interval specified by the *time\_spec* argument.

```
ncsim> strobe -time 100ns x y z
ncsim> strobe -time 100ns x {y %b} z
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

#### **-delete**

Deletes the strobe.

#### **-help**

Displays help on the strobe command.

You cannot use the `help` command to display help on the `strobe` command because it is a Tcl procedure. The `help` command only displays help on simulator commands that are implemented in C.

## **strobe Command Examples**

The following command prints the values of signals `clk`, `clr`, `data`, and `q` every 100 ns.

```
ncsim> strobe -time 100 clk clr data q
Setting up strobe time - '100'
ncsim> run
Time          |clk  |clr   |data  |q    |
-----
100 NS        |1'h1 |1'h1  |4'h0  |4'hx |
200 NS        |1'h1 |1'h1  |4'h1  |4'h0 |
300 NS        |1'h1 |1'h1  |4'h2  |4'h1 |
400 NS        |1'h1 |1'h1  |4'h3  |4'h2 |
...

```

The following command prints the values of signals `clk`, `clr`, `data` (in binary), and `q` every 100 ns.

```
ncsim> strobe -time 100ns clk clr {data %b} q
Setting up strobe time - '100ns'
ncsim> run 300 ns
Time          |clk  |clr   |data    |q    |
-----
100 NS        |1'h1 |1'h1  |4'b0000 |4'hx |
200 NS        |1'h1 |1'h1  |4'b0001 |4'h0 |
300 NS        |1'h1 |1'h1  |4'b0010 |4'h1 |
Ran until 300 NS + 0

```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

In the following command sequence, the simulation is run for 300 ns. When the simulation is resumed, the strobe header is not displayed. The `strobeHeader` variable is then set to display the header when the simulation is resumed.

```
ncsim> strobe -time 100 clk clr data q
Setting up strobe time - '100'
ncsim> run 300
Time |clk |clr |data |q |
-----
100 NS |1'h1 |1'h1 |4'h0 |4'hx |
200 NS |1'h1 |1'h1 |4'h1 |4'h0 |
300 NS |1'h1 |1'h1 |4'h2 |4'h1 |
Ran until 300 NS + 0
ncsim> run 300
400 NS |1'h1 |1'h1 |4'h3 |4'h2 |
500 NS |1'h1 |1'h1 |4'h4 |4'h3 |
600 NS |1'h1 |1'h1 |4'h5 |4'h4 |
Ran until 600 NS + 0
ncsim> set strobeHeader 1
1
ncsim> run 300
Time |clk |clr |data |q |
-----
700 NS |1'h1 |1'h1 |4'h6 |4'h5 |
800 NS |1'h1 |1'h1 |4'h7 |4'h6 |
900 NS |1'h1 |1'h1 |4'h8 |4'h7 |
Ran until 900 NS + 0
```

To change the format globally, you can set the `strobeFmt` variable, as shown in the following example. You must reset the strobe after setting the `strobeFmt` variable.

```
ncsim> strobe -time 100 clk clr data q
Setting up strobe time - '100'
ncsim> run 300
Time |clk |clr |data |q |
-----
100 NS |1'h1 |1'h1 |4'h0 |4'hx |
200 NS |1'h1 |1'h1 |4'h1 |4'h0 |
300 NS |1'h1 |1'h1 |4'h2 |4'h1 |
Ran until 300 NS + 0
ncsim> set strobeFmt %b
%b
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
ncsim> strobe -time 100 clk clr data q
Setting up strobe time - '100'
ncsim> run 300
Time          |clk  |clr   |data    |q      |
-----
400 NS        |1'b1 |1'b1  |4'b0011 |4'b0010 |
500 NS        |1'b1 |1'b1  |4'b0100 |4'b0011 |
600 NS        |1'b1 |1'b1  |4'b0101 |4'b0100 |
Ran until 600 NS + 0
```

The following command displays the values of `clk`, `clr`, and `q` when `data` changes value.

```
ncsim> strobe -object data clk clr q
Setting up strobe object - 'data'
ncsim> run 300 ns
Time          |clk  |clr   |q      |
-----
0 NS         |1'h0 |1'h1  |4'hx  |
100 NS        |1'h1 |1'h1  |4'hx  |
200 NS        |1'h1 |1'h1  |4'h0  |
Ran until 300 NS + 0
```

The following command displays the values of `data`, `clk`, `clr`, and `q` when `data` changes value.

```
ncsim> strobe -object data data clk clr q
Setting up strobe object - 'data'
ncsim> run 300 ns
Time          |data |clk  |clr   |q      |
-----
0 NS         |4'h0 |1'h0  |1'h1  |4'hx  |
100 NS        |4'h1 |1'h1  |1'h1  |4'hx  |
200 NS        |4'h2 |1'h1  |1'h1  |4'h0  |
Ran until 300 NS + 0
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following command displays the values of data (in binary), clk, clr, and q (in hex) when data changes value. Notice that the object-format pair is enclosed in curly braces.

```
ncsim> strobe -object data {data %b} clk clr q
Setting up strobe object - 'data'
ncsim> run 300 ns
Time          |data    |clk    |clr    |q     |
-----
0 NS          |4'b0000 |1'h0   |1'h1   |4'hx  |
100 NS         |4'b0001 |1'h1   |1'h1   |4'hx  |
200 NS         |4'b0010 |1'h1   |1'h1   |4'h0  |
Ran until 300 NS + 0
```

The following command displays the values of the specified signals when data has the value 3, decimal. The signal data is available from the top level of the hierarchy.

Verilog:

```
ncsim> strobe -condition {[value %d top.data] = 3} clk clr data q
```

VHDL:

```
ncsim> strobe -condition {[value %d :data] = 3} clk clr data q
```

If you are currently at the top level, you can omit the hierarchical path specification to data, and write the two commands shown in the previous example as follows:

```
ncsim> strobe -condition {[value %d data] = 3} clk clr data q
```

Instead of using the value command to get the value of data into the expression evaluator, you can use #data. Include the format specifier after the # sign.

```
ncsim> strobe -condition {#%ddata = 3} clk clr data q
```

The following command displays the values of the specified signals when data has the value 0010 (binary). For VHDL, you must enclose vectors in quotation marks.

Verilog:

```
ncsim> strobe -condition {#data = 4'b0010} clk clr data q
```

VHDL:

```
ncsim> strobe -condition {#data ="0010"} clk clr data q
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following command displays the values of the specified signals when bit 0 of data is 1. The expression is evaluated when any bit of data changes value. For VHDL, you must enclose single-bit entities in single quotation marks.

Verilog:

```
ncsim> strobe -condition {#data[0] == 1} clk clr data q
```

VHDL:

```
ncsim> strobe -condition {#data(0) == '1'} clk clr data q
```

The following command is identical to the previous command except that it uses an explicit `value` command to get the value of data (bit 0) into the expression parser.

Verilog:

```
ncsim> strobe -condition {[value %b data[0]] == 1'b1} clk clr data q
```

VHDL:

```
ncsim> strobe -condition {[value %b data(0)] == '1'} clk clr data q
```

The following command displays the values of the specified signals if the value of `top.load` has the value `x`. Notice that in the Tcl expression, the case-equality operator (`==`) is used. For this operator, bits that are unknown are included in the comparison, and the result of the expression is always 1 (true) or 0 (false).

```
ncsim> strobe -condition {#top.load === 1'bx} clk clr data q
```

In the following command, the logical comparison operator (`=` or `==`) is used. These operators return the unknown value (`x`) if either operand is unknown. In a conditional expression, an unknown result is treated as false. Therefore, the following command does *not* stop the simulation when the signal has the value `x`.

```
ncsim> strobe -condition {#top.load == 1'bx} clk clr data q
```

## task

The `task` command lets you schedule Verilog tasks for execution. Verilog tasks are those tasks that are implemented in the Verilog source code with a `task` declaration. You cannot use the `task` command to schedule the execution of built-in or user-defined Verilog system tasks, Verilog functions, VHDL functions, or VHDL procedures.

The `task` command schedules the specified task(s) to execute at the current simulation time. However, the task is not executed immediately (as is the case in Verilog-XL). The task begins executing after you resume the simulation, and is executed at some point during the current simulation time. The simulator may execute other behaviors that are already scheduled for the current simulation time before executing the scheduled task.

When scheduling tasks that may be called elsewhere (from within the Verilog code, from VPI, or from Tcl), you must remember that, because tasks in Verilog are static, multiple concurrent calls to tasks can interfere with each other, stomping on the values of parameters and registers within the task.

Parameters to tasks are static registers. You must set the input and inout parameters before the scheduled task runs by using a `deposit` command. You can schedule the task before or after you deposit values to the input and inout parameters. Remember that intervening calls to the same task in the model can interfere with and overwrite the values that you deposited to the task parameters before the scheduled task actually runs.

You can read the value of output parameters after the task has completed by using the `value` command.

**Note:** To deposit values to parameters and to read output parameters, the parameters must have read/write access. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details on specifying access to simulation objects.

Disable statements in the Verilog code affect active tasks that you scheduled with the `task` command in the same way that they affect tasks that are scheduled by the code in the model.

Once a task is scheduled, it becomes part of the model’s state. If you save a snapshot with a task that is scheduled or that is in progress, the task will still be scheduled or in progress when you restart that snapshot. Tasks that are scheduled or that are in progress at the time of the save will be the only tasks that are scheduled or in progress after the restart.

### task Command Syntax

```
task [-schedule] task_name [task_name ...]
```

## task Command Modifiers and Options

### **[-schedule] task\_name [task\_name ...]**

Schedule the specified task(s) for execution at the current simulation time.

The task begins executing after the simulation is resumed, and at some point during the current simulation time. Other behaviors that are already scheduled for the current simulation time may execute before the scheduled task starts.

The `-schedule` modifier is optional.

## task Command Examples

The following source code is used in the examples in this section:

```
module top;
reg out;
reg i1,i2,i3;
reg clock;
initial
    #10000 $display("DONE");
always
    @(posedge clock) print_task1(out,i1,i2,i3);

task print_task1;
    output [31:0] out;
    input [31:0] i1;
    input [31:0] i2;
    input [31:0] i3;
begin
    out = i1 + i2 + i3;
    $display("%t %0d %0d %0d %0d", $time,out,i1,i2,i3);
end
endtask

task print_task2;
    output [31:0] out;
    input [31:0] i1;
    input [31:0] i2;
    input [31:0] i3;
begin
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
    out = i1 * i2 * i3;
    $display( "%t %0d %0d %0d %0d", $time, out, i1, i2, i3 );
end
endtask
endmodule
```

The following sequence of commands schedules the task called `print_task1` for execution and sets the input parameter values that will be in effect when the task runs. The `run` command then continues the simulation so that the task is executed.

```
ncsim> run 50
ncsim> task -schedule print_task1
ncsim> deposit print_task1.i1 1
ncsim> deposit print_task1.i2 1
ncsim> deposit print_task1.i3 1
ncsim> run 1
      50 3 1 1 1
Ran until 51 NS + 0
```

The following command specifies the full path to the task. The `-schedule` modifier is not required.

```
ncsim> task top.print_task1
```

To schedule more than one task, specify the task names on the command line and then deposit values to the input and inout parameters.

```
ncsim> task print_task1 print_task2
ncsim> deposit print_task1.i1 1
ncsim> deposit print_task1.i2 1
ncsim> deposit print_task1.i3 1
ncsim> deposit print_task2.i1 1
ncsim> deposit print_task2.i2 1
ncsim> deposit print_task2.i3 1
ncsim> run 1
      0 3 1 1 1
      0 1 1 1 1
Ran until 1 NS + 0
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

If you want to schedule a task at a specific time, set a time breakpoint, advance the simulation to that time, and then schedule the task.

```
ncsim> stop -time 50 -absolute  
ncsim> run  
ncsim> task print_task2  
ncsim> (deposit values to parameters)  
ncsim> run
```

In the following sequence of commands, a line breakpoint is set on the first statement in the task so that the simulation stops when the task is about to execute. Parameter values are then deposited immediately before the execution of the task.

```
ncsim> stop -line 18  
ncsim> run  
ncsim> deposit i1 0  
ncsim> deposit i2 1  
ncsim> deposit i3 1  
ncsim> run 1  
    100 2 0 1 1  
Ran until 101 NS + 0
```

In the following sequence of commands, a line breakpoint is set on the first statement in the task. After depositing values to the input parameters, a `run -return` command is issued. This stops the simulation when the task returns so that you can read output parameters as soon as they are available.

```
ncsim> stop -line 18  
Created stop 1  
ncsim> run  
100 NS + 0 (stop 1: ./test2.v:18)  
. ./test2.v:18      begin  
ncsim> deposit i1 1  
ncsim> deposit i2 0  
ncsim> deposit i3 0  
ncsim> run -return  
    100 1 1 0 0  
. ./test2.v:10 @(posedge clock) print_task1(out,i1,i2,i3);
```

## time

The `time` command displays the current simulation time scaled to the specified unit. The unit can be:

- a time unit that you specify
- `auto`—use the largest base unit that makes the numeric part of the time an integer
- `module`—use the timescale of the current debug scope

The simulation time can be displayed in the following time units:

- `fs`—femtoseconds
- `ps`—picoseconds
- `ns`—nanoseconds
- `us`—microseconds
- `ms`—milliseconds
- `sec`—seconds

If no unit is given, the value of the `$display_unit` variable is used. This variable is set to `auto` by default.

You also can display the time in 10 or 100 times the base unit. For example,

```
ncsim> time fs  
ncsim> time 10fs  
ncsim> time 100fs
```

## time Command Syntax

```
time [[10 | 100]time_unit | auto | module]  
      -delta  
      -nounit
```

## time Command Modifiers and Options

### **-delta**

Includes the delta cycle count.

At any given simulation time, values of nets are first updated and then behaviors that are sensitive to those nets are executed. This two step process may be repeated any number of times because of zero-delays. The delta cycle count represents the number of times the process is repeated for the given simulation time.

### **-nounit**

Does not include the time unit.

## time Command Examples

```
% ncsim -tcl board
ncsim: v1.0.(p2): (c) Copyright 1995, 1996 Cadence Design Systems, Inc.
ncsim> run 100 ns
5  count= X, f=x, af=x
Ran until 100 NS + 0
```

The following command displays the current simulation time in ns.

```
ncsim> time ns
100 NS
```

The following command displays the current simulation time in fs.

```
ncsim> time fs
100000000 FS
```

The following command displays the current simulation time in 100 times the base unit of fs.

```
ncsim> time 100fs
1000000 100FS
```

The following commands illustrate the `auto` keyword, which displays the time using the largest base unit that makes the numeric part of the time an integer.

```
ncsim> time fs
100000000 FS
ncsim> time auto
100 NS
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following command displays the current simulation time using the timescale of the current debug scope.

```
ncsim> time module
100 NS
```

The following command displays the current simulation time using the timescale of the current debug scope and including the delta cycle count.

```
ncsim> time module -delta
100 NS + 0
```

The following command displays the current simulation time with no time unit.

```
ncsim> time -nounit
100
```

## **value**

The `value` command prints the current value of the specified objects using the last format specifier preceding the object name argument. If no format is specified, a default format is used.

Objects specified as arguments to the `value` command must have read access. An error is printed if an object does not have read access.

### **value Command Syntax**

```
value [format ...] object_name ...
```

For Verilog, the valid formats are:

<b>Format Specifier</b>	<b>Format</b>
%c	character
%s	string
%b	binary
%d	decimal
%o	octal
%x	unsigned hexadecimal
%h	same as %x
%f	floating-point number
%e	real number in mantissa-exponent form
%g	use %e or %f, whichever is shorter
%t	decimal time scaled from the timescale of the object's module to the simulation's timescale
%v	strength value—wires only

To revert to the default format, use %.

If no format is specified, the default format depends on the object type. The following defaults are used:

- time—%d
- integer—%d
- real—%g
- reg—\$vlog\_format
- wire—\$vlog\_format

where `$vlog_format` is a predefined Tcl variable that defaults to %h. You can set this variable to %b, %o, or %d.

For VHDL, values are returned in a format that resembles the appropriate VHDL syntax for the object type. If one of the radix format specifiers (%b, %o, %d, or %x) is given, the format affects the format of integer values and of bit\_vector and std\_logic\_vector values. Otherwise, the format specifier is ignored for VHDL values.

You can use wildcard characters in the argument to a `value` command.

- The asterisk ( \* ) matches any number of characters.
- The question mark ( ? ) matches any one character.

You cannot use wildcard characters inside escaped names.

See “[Using Wildcards Characters in Tcl Commands](#)” on page 495 for more information on using wildcards.

## **value Command Modifiers and Options**

None.

## value Command Examples

The following command displays the value of the signal `data`.

```
ncsim> value top.u1.data  
4'h2
```

If the current debug scope has been set to instance `u1`, you can display the value of `data` with the following command.

```
ncsim> value data  
4'h2
```

The following command displays the value of all objects in the current scope that have names that start with the letter `c`.

```
ncsim> value c*  
4'ha 1'h0
```

The following command displays the value of all objects in the current scope that have names that are six characters long and that start with `rst` and end with `p5`.

```
ncsim> value rst?p5  
1'h0 1'h0 1'h0
```

The following sequence of `value` commands displays the current value of `data` in a variety of formats.

```
ncsim> value data  
4'h2  
ncsim> value %o data  
4'o02  
ncsim> value %b data  
4'b0010  
ncsim> value %d data  
4'd2  
ncsim> value %g data  
2  
ncsim> value %f data  
2.000000  
ncsim> value %e data  
2.000000e+00  
ncsim> value %b data %d q  
4'b0010 4'd1  
ncsim> value % data %d data %b data  
4'h2 4'd2 4'b0010
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

The following command shows the error message that is displayed when you run in regression mode (no read, write, or connectivity access to simulation objects) and then use the **value** command on an object that does not have read access.

```
ncsim> value clk
ncsim: *E,RDACRQ: Object does not have read access: hardrive.clk.
```

In the following VHDL example, the **stack -show** command displays the current subprogram call stack. The process (**process1**) is displayed as nest-level 0, the base of the stack. The subprogram **function1** is **:process1[1]**, and the subprogram **function2** is **:process1[2]**.

- The first **value** command displays the value of the variable **tmp5** in the current debug scope.
- The second **value** command displays the value of the variable **var1** in **:process1**.
- The third **value** command displays the value of the signal **tmp4** in **:process1[1]** (that is, in **function1**).
- A **scope** command is then executed to set the scope to **function1**. The last **value** command displays the value of **tmp4**.

```
ncsim> stop -subprogram function1
Created stop 1
ncsim> run
0 FS + 0 (stop 1: Subprogram :function1)
./test.vhd:36 tmp4_local := function2 (tmp4);
ncsim> run -step
./test.vhd:29 tmp5_local := tmp5 + 1;
ncsim> stack -show          ;# Display the current call stack
2: Scope: :process1[2] Subprogram:@work.e(a):function2
   File: /usr1/belanger/inca/vhdl/subprogram_debug	scope_test/test.vhd
   Line: 29

1: Scope: :process1[1] Subprogram:@work.e(a):function1
   File: /usr1/belanger/inca/vhdl/subprogram_debug	scope_test/test.vhd
   Line: 36

0: Scope: :process1
   File: /usr1/belanger/inca/vhdl/subprogram_debug	scope_test/test.vhd
   Line: 52
```

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

```
ncsim> scope -show                      ;# Display the current debug scope
Directory of scopes at current scope level:

Current scope is (:process1[2])
Highest level modules:
    Top level VHDL design unit:
        entity (e:a)
    VHDL Package:
        STANDARD
        ATTRIBUTES
        std_logic_1164
        TEXTIO
ncsim> value tmp5                      ;# Display the value of tmp5 in :process1[2]
                                              ;# i.e., function2
1
ncsim> value :process1:var1            ;# Display the value of var1 in :process1
'0'
ncsim> value :process1[1]:tmp4          ;# Display the value of tmp4 in :process1[1]
                                              ;# i.e., function1
1
ncsim> scope -set :process1[1]          ;# Set scope to function1 (:process1[1])
ncsim> value tmp4                      ;# Display the value of tmp4
1
```

## **version**

The **version** command displays the version number of *ncsim*.

You can also print the version number by invoking the simulator with the **-version** option.

```
% ncsim -version
```

### **version Command Syntax**

```
version
```

### **version Command Modifiers and Options**

None.

### **version Command Examples**

```
ncsim> version
ncsim: v3.0.(p1)
```

## **where**

The **where** command displays the current location of the simulation. This includes the current simulation time and the current scope.

### **where Command Syntax**

`where`

### **where Command Modifiers and Options**

None.

### **where Command Examples**

```
ncsim> where
TIME: 3400 NS + 0
Scope is (board.counter)
ncsim>
```

```
ncsim> where
TIME: 100 NS + 0
Scope is (:top.VENDING)
ncsim>
```

# Verilog-XL and NC-Verilog Simulator Interactive Debug Commands

---

## Simulation Control Commands

Description	XL Command	Tcl Command
Run	.	run
Run 100 ns	#100 \$stop;	run 100 ns
Run 1 event	;	run -step
Exit simulator	\$finish;	exit or finish
Save simulation	\$save("fname");	save <i>cellname</i>
Restore simulation	\$restart("fname");	restore <i>cellname</i>
Reset to time 0	\$reset;	reset

---

## Breakpoint Commands

Description	XL Command	Tcl Command
Relative time breakpoint	#100 \$stop;	stop -time 100 ns -delbreak 1
Absolute time breakpoint	\$db_breakbeforetime(100);	stop -time 100 ns -absolute
Object breakpoint	@(clk) \$stop; or \$db_breakoncwhen (clk);	stop -object clk -delbreak 1
Condition breakpoint	wait (A&B) \$stop;	stop -condition {#A & #B} -delbreak 1
Source Line breakpoint	\$db_breakoncitatline(23);	stop -line 23 -delbreak 1

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

#### **Breakpoint Commands**

Description	XL Command	Tcl Command
Show active breakpoint	\$history; (* denotes active commands)  To display \$db breakpoints:  \$db_showbreak;	stop -show
Disable or delete breakpoints	-interactive_cmd_num  or if a \$db breakpoint:  \$db_disablebreak (break_num);  \$db_deletebreak (break_num);	stop -delete break_num  stop -disable break_num
Enabling breakpoint	\$db_enablebreak (break_num);	stop -enable break_num

---

#### **Simulation Debug Commands**

Description	XL Command	Tcl Command
Force identifier to a value	force clk = 1;	force clk = 1
Release forced identifier	release clk;	release clk
Deposit value on identifier	\$deposit(clk, 1);	deposit clk = 1
Display current simulation time	\$display(\$time);	time or time -ns
Display drivers of net or reg	\$showvars(clk);	drivers clk
Display value of object	\$display(clk);	value clk
Display value of object formatted	\$display("clk = %b", clk);	puts "clk = [value %b clk]"
Display value of object w/o carriage return	\$write("clk = %b", clk);	puts -nonewline "clk = [value %b clk]"

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

#### **Simulation Debug Commands**

Description	XL Command	Tcl Command
Display value of all variables in current scope	\$showvariables;	scope -drivers
Display instances in current scope	\$showscopes;	scope -show
Display current debug scope	:	scope
Set debug scope	\$scope(A.B.C);	scope A.B.C
Set debug scope up one level		scope -up

---

#### **VCD Waveform Commands**

Description	XL Command	Tcl Command
Open VCD file	\$dumpfile ( "fname.vcd" );	database -default -vcd <i>fname</i> -into <i>fname.vcd</i>
Add signal to VCD file	\$dumpvars(0, <i>signame</i> );	probe <i>signame</i> -vcd
Add all signals in an instance to VCD file	\$dumpvars(1, <i>instname</i> );	probe <i>instname</i> -all -vcd
Add all signals in an instance and all subinstances to VCD file	\$dumpvars(0, <i>instname</i> );	probe <i>instname</i> -all -depth all -vcd
Add all ports in an instance to VCD file		probe <i>instname</i> -ports -vcd
Add all ports in an instance and all subinstances to VCD file		probe <i>instname</i> -ports -depth all -vcd
Show probes		probe -show
Delete probe		probe -delete <i>probename</i>

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

#### VCD Waveform Commands

Description	XL Command	Tcl Command
Show open databases		database -show
Disable waveform capture	\$dumpoff ;	database -disable <i>fname</i>
Enable waveform capture	\$dumpon ;	database -enable <i>fname</i>

---

#### SHM Waveform Commands

Description	XL Command	Tcl Command
Open SHM database	\$shm_open ( "fname.shm" );	database -default -shm <i>fname</i> -into <i>fname.shm</i>
Add signal to SHM database	\$shm_probe( <i>signame</i> );	probe <i>signame</i> -shm
Add all signals in an instance to SHM database	\$shm_probe( "A" , <i>instname</i> );	probe <i>instname</i> -all -shm
Add all signals in an instance and all subinstances to SHM database	\$shm_probe( "AC" , <i>instname</i> );	probe <i>instname</i> -all -depth all -shm
Add all ports in an instance to SHM database	\$shm_probe( <i>instname</i> );	probe <i>instname</i> -ports -shm
Add all ports in an instance and all subinstances to SHM database	\$shm_probe( "C" , <i>instname</i> );	probe <i>instname</i> -ports -depth all -shm
Show probes		probe -show
Disable probe		probe -disable <i>probename</i>

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

#### **SHM Waveform Commands**

Description	XL Command	Tcl Command
Enable probe		probe -enable <i>probename</i>
Delete probe		probe -delete <i>probename</i>
Show open databases		database -show
Disable waveform capture		database -disable <i>fname</i>
Enable waveform capture		database -enable <i>fname</i>
Close SHM database	\$shm_close;	database -close <i>fname</i>

---

#### **Other Commands**

Description	XL Command	Tcl Command
Command alias	`define h \$history;	alias h history
Command history	\$history;	history
Replay previous command	<i>cmd_number</i>	! <i>cmd_number</i>
Save tcl enviornment		save -environment <i>fname</i>
Load file of interactive commands	\$input("fname");	source <i>fname</i>
Decompile Verilog source	\$list( <i>instname</i> );	scope -list <i>instname</i>
Display message to log file	\$display("Hello World");	puts "Hello World"
Create additional log file		set fileid [ open "logfilename" w]
Display message to alternate log file		puts \$fileid "clk = [value clk]"

## NC-Verilog Simulator Help

### Using the Tcl Command-Line Interface

---

---

#### Other Commands

Description	XL Command	Tcl Command
Help		help
Help on a specific command		help <i>command</i>

---

## Using the SimVision Analysis Environment

---

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

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

- The SimControl window, which allows you to directly interact with the simulator. You can single step, trace signals, set breakpoints, and observe signals to verify your designs. SimControl also includes the following debug tools:
  - ❑ The Watch Objects Window, which lets you observe the value of selected signals.
  - ❑ The Navigator, which lets you view the design hierarchy and which shows you signal values at any level of a design hierarchy.
  - ❑ The Signal Flow Browser, which lets you trace signals to find the source of a problem.
  - ❑ The Cycle View , which lets you step through the simulation cycle to debug delta cycle bugs in the design.
  - ❑ The Schematic Window, which lets you view RTL models in a schematic form. The Schematic Window can display the schematic for an entire module that you select in the Navigator, or it can display a signal and its connected logic if you select a signal in the Source Browser or other SimControl tool.

In the current release, the Schematic Window is available only for pure Verilog designs. This tool is not available in the Verilog desktop simulator.

- SimVision Waveform Viewer—Lets you display waveforms.
- Comparescan—Lets you compare SHM and VCD waveform databases.

## NC-Verilog Simulator Help

### Using the SimVision Analysis Environment

---

**Note:** The Cycle View, Schematic Window, and Comparescan are not currently available on Windows platforms.

You can invoke the SimVision debug environment in post-processing mode. The SimVision Post Processing Environment (PPE) lets you analyze simulation results stored in an SHM database and debug your design without using a simulator license. The PPE gives you access to all SimVision tools available in interactive mode, and the features of each tool are virtually identical to those in interactive mode.

On UNIX, you invoke the simulator with the SimVision environment by including the `-gui` option on the command line (`+gui` if you are running in single-step invocation mode with the `ncverilog` command).

```
% ncsim -gui snapshot_name  
% ncverilog +gui source_files
```

On Windows, you invoke the simulator from NCLaunch by selecting *Tools—Simulator* or by clicking the *Launch Simulator* button on the tool bar. The `-gui` (`+gui`) option is selected by default, so the simulator will be invoked with SimVision automatically.

See the [\*Cadence SimVision Analysis Environment\*](#) documentation for details on using the debug environment.

For an introduction to using the SimVision analysis environment, see the [\*Cadence NC-Verilog Simulator Tutorial\*](#).

---

## Maximizing Simulation Performance

---

This chapter discusses ways to maximize simulation performance using the NC-Verilog simulator. This chapter contains the following sections:

- Coding Style Guidelines
  - General Guidelines
  - Recommended Verilog Coding Practices
  - Coding Styles to Avoid
- Refining the Testbench Strategy
- Avoiding Unnecessary Recompilation
- Using the Profiler to Identify and Eliminate Simulation Bottlenecks

## Coding Style Guidelines

Perhaps the most important factor in simulation performance is the code that is being simulated and how it is written. There are usually many ways to model specific pieces of hardware, to write a PLI application, and to write and apply stimulus. For an event-driven simulator, such as NC-Verilog, some coding styles are more efficient in simulation than others, due to differences in how many events they create. For example, adding unnecessary assignments or gates will cause any event-driven simulator to simulate more events, and thus do more work than necessary. In addition, some coding styles are more efficient than others because they allow the simulator to apply algorithms that help it to accelerate the simulation.

This section is divided into three subsections:

- General Guidelines

Presents some general, high-level strategies for achieving optimal simulation performance.

- Recommended Verilog Coding Practices

Presents Verilog coding styles that will simulate with the best performance.

- Coding Styles to Avoid

Shows you examples of Verilog code that will slow down simulation.

### General Guidelines

This section presents some high-level strategies for achieving optimal simulation performance.

#### Simulate with Code Written at Higher Levels of Abstraction

Like most simulators, NC-Verilog simulates behavioral/RTL code faster than it simulates gate-level or switch-level designs.

For regression testing, it is best to stay away from gates and switches if possible, and simulate with code written at higher levels of abstraction, such as RTL or behavioral. In particular, designs coded with RTL representations of flip-flops tend to simulate much faster than those with gate-level or switch-level representations.

## Waveform Dumping

Dumping waveforms has a high impact on simulator performance. Avoid dumping waveforms for all tests in a regression run. Most regression tests pass and you will not need the waveforms dumped for these tests. Run all of the tests with no waveform dumping and then, if necessary, simulate the failed tests a second time, dumping waveforms. In other words, do not dump waveforms just because you think that you *might* need them later.

## Avoid Unnecessary File I/O

One of the biggest bottlenecks in designs and testbenches is excessive file I/O. While writing to files, displaying messages, dumping waveforms, and so on, are all useful techniques for debugging, they may be unnecessarily excessive. For example, displaying status messages to the screen or log file will create a bottleneck for the simulation. The purpose of this I/O might be to help locate where errors occur, but this dumping of data is often inefficient and avoidable.

The goal of regression testing is to run as many tests as possible to find the few bugs that are left. True regression runs are self-checking and should require only minimal screen and file I/O. Should one of the tests find a bug and cause that test to fail, that particular test can be rerun with waveforms and informative messages.

## Improve Inefficient PLI

Another common bottleneck in designs is PLI. In some cases, the culprit is a third-party tool that you have no control over. However, if you do have control over the PLI code, you should review this code and try to streamline it. Common inefficiencies in PLI are too much file I/O, too many unnecessary accesses to the simulation's data structure, and inefficient algorithms in the code.

## Recommended Verilog Coding Practices

This section presents some coding guidelines that can increase the performance of the NC-Verilog simulator. It tells you how to write Verilog code that will simulate with the best performance. Because designers have the most coding flexibility and the most opportunities for tuning the code for simulation speed when writing procedural RTL, most of the focus in the following sections is on procedural RTL, particularly some of the most commonly-used pieces of code, such as flip-flops and stimulus.

Most synchronous designs spend much of their time simulating simple flip-flops, which are the baseline of synchronous design. NC-Verilog includes algorithms that optimize the performance of simple, properly-coded flip-flops. There are rules to which the flip-flop must

conform in order for these optimizations to be performed, and the next few pages outline how to properly code flops that will be optimized by the simulator.

## Simple DFF

The base element of most synchronous design is the D flip-flop (DFF). The following figure illustrates how to code a simple one-bit wide RTL DFF:

### Good:

```
module DFF (q, d, clk);
output q;
input d, clk;
reg q;

always @(posedge clk)
  q <= d;
endmodule
```

### Bad:

```
module DFF (q, d, clk);
output q;
input d, clk;
reg q;

always @(posedge clk)
begin
  if (clk !== 1'bX)
    $display ("ERROR");
  else q = #1 d;
end
endmodule
```

### Guidelines:

- Use non-blocking procedural assignments (`<=`). This type of assignment is just as fast as a blocking assignment (`=`), and is generally preferred to avoid race-conditions, or event-ordering dependencies within a time slice.
- Avoid blocking assignments with a delay, such as the following:  
`q = #1 d;`  
This type of assignment disables optimizations.
- Make sure that the action is independent of the `clk` value.
- Do not use system tasks in an `always` block. This disables optimizations.

## N-Bit Wide DFF

To design registers that are wider than one bit, it is more efficient to create an n-bit representation of the DFF shown in the previous section than it is to string *n* number of these flops together, because performance is more dependent on the number of operations than on the size of the operations. For example, a single 32-bit operation is much faster than 32 single-bit operations.

For flexibility purposes, you can create a DFF model with a parameterized width. When using this model, the width parameter can be overridden at the instantiation with the appropriate value. For example:

```
module DFF (q, d, clk);
parameter WIDTH = 8;
output [WIDTH - 1 : 0] q;
input [WIDTH - 1 : 0] d;
input clk;
reg [WIDTH - 1 : 0] q;

always @(posedge clk)
  q <= d;

endmodule
```

Instantiations of this flop might look like the following, which represent a two-bit data bus and a 32-bit data bus.

```
DFF #(2) DFF_inst_0 (out0, in0, clk);
DFF #(32) DFF_inst_1 (out1, in1, clk);
```

This style not only allows the n-bit wide register to simulate faster, but also allows the simulator to keep the input and output vectors as one unit, rather than having to expand them to individual bits.

Guidelines:

- Create an n-bit representation of the DFF rather than stringing together *n* number of one-bit DFFs.
- Leave buses intact. Avoid bit-selects (`foo[3]`) or part-selects (`foo[5:3]`).
- Don't force expansion by using the `ncelab -expand` (`ncverilog +ncexpand`) option.

**Note:** To avoid vector expansion, it is important to remember that a vector will be expanded to individual bits if there is any point along its path where it must be expanded. For more on vector expansion, see ["Avoid Unnecessary Vector Expansion" on page 687](#).

## DFF With Asynchronous Reset

You can add an asynchronous active-low reset signal to this DFF model in two ways, both of which allow the DFF to be optimized. Note, however, that in order for a DFF to qualify for optimization, all assignments to its regs must be of the same type. In other words, if `q` is

assigned in one place with a non-blocking procedural assignment, it must be assigned everywhere with one.

The first example uses multiple `always` blocks.

```
module DFF (q, d, clk, rst);
parameter WIDTH = 8;
output [WIDTH - 1 : 0] q;
input [WIDTH - 1 : 0] d;
input clk, rst;
reg [WIDTH - 1 : 0] q;

always @(posedge clk)
  if (rst)
    q <= d;

always @(negedge rst)
  q <= 0;

endmodule
```

This reset modeling style is cycle-simulatable. However, some synthesis tools prefer that the clock and reset be contained in one `always` block, as shown in the following example:

```
module DFF (q, d, clk, rst);
parameter WIDTH = 8;
output [WIDTH - 1 : 0] q;
input [WIDTH - 1 : 0] d;
input clk, rst;
reg [WIDTH - 1 : 0] q;

always @(posedge clk or negedge rst)
  if (!rst)
    q <= 0;
  else
    q <= d;

endmodule
```

This second model will simulate as fast as the previous example as long as every flop that is driven by the clock signal is also dependent on the same reset signal. This can be hard to keep track of in a real design if there is one clock signal with a large amount of fanout that goes to all kinds of blocks, which may or may not have the same reset line, if they have a reset at all. In this case, it can be worthwhile to have separate simulation and synthesis versions of this module. You can accomplish this either with an `'ifdef` conditional compile, or by having separate files, either in different directories or with different filename extensions.

Guidelines:

- All assignments to regs of the DFF must be of the same type. Do not mix blocking and non-blocking assignments.
- The second example, with a single `always` block, may be preferred by the synthesis tool. However, every flop that is driven by the clock signal must also be dependent on the same reset signal.

### Delays in RTL Code

Adding delays to RTL code is a common practice that is used either to add some timing to the RTL simulation, or to just offset the output transition from the clock for readability in a waveform display.

To model delays with non-blocking assignments in the DFF, the delay values can be added to the assigning statements themselves. Every assignment to the `reg` must be delayed by the same amount in order to take advantage of the optimizations.

This method takes away the ability to specify different rise and fall delays. However, this method is the simplest, and is more generally accepted by synthesis tools, which would probably eliminate the delay anyway.

Here is an example using the synthesizable DFF presented in the preceding section:

```
module DFF (q, d, clk, rst);
parameter WIDTH = 8;
parameter DELAY = 1;
output [WIDTH - 1 : 0] q;
reg [WIDTH - 1 : 0] q;
input [WIDTH - 1 : 0] d;
input clk, rst;

always @(posedge clk or negedge rst)
  if (!rst)
    q <= #(DELAY) 0;
  else
    q <= #(DELAY) d;

endmodule
```

You could also model the DFF using a continuous assignment with some delay to the output port, as shown in the following example. In this example, the use of non-blocking assignments is not necessary, since the delayed continuous assignment will prevent any possible race conditions.

```
module DFF (q, d, clk, rst);
parameter WIDTH = 8;
parameter DELAY = 1;
output [WIDTH - 1 : 0] q;
input [WIDTH - 1 : 0] d;
input clk, rst;
reg [WIDTH - 1 : 0] q_reg;

assign #(DELAY) q = q_reg;

always @(posedge clk)
  if (rst)
    q_reg = d;

always @(negedge rst)
  q_reg = 0;

endmodule
```

**Guidelines:**

- For both styles shown above, all assignments must use the same delay.
- Do not mix blocking and non-blocking assignments.

### DFF Modeled with a User-Defined Primitive

Various constraints may require that you use user-defined primitives (UDP's) rather than RTL flip-flops. NC-Verilog can optimize UDP's so that they simulate very fast. In fact, for a one-bit-wide flop, a UDP can actually be faster than its one-bit-wide always block RTL counterpart.

This difference in speed is an example of the effect of dealing with a construct more than once during an evaluation. When the clock input to the UDP changes, it is evaluated immediately, so it is dealt with only once. On the other hand, when the clock input to the always block changes, it is scheduled to be evaluated later, so it is dealt with twice. If the always block uses a non-blocking assignment to set its output, it must be dealt with again to finish the assignment at the end of the time slice.

There are two restrictions on modeling UDPs so that they qualify for optimization:

- State transitions based on non-clock signals must be independent of the clock value. For these lines, use the question mark symbol ( ? ) for the clock.

- No state transitions are permitted on the non-active edge of the clock. For these lines, use a dash ( - ).

The following example illustrates the restrictions:

```
primitive DFF (q, d, clk, rst);
output q;
input d, clk, rst;
reg q;

table
// d  clk  rst  :  q   :  next_q
 0  r    1    :  ?  :    0 ;
 1  r    1    :  ?  :    1 ;
  ?  ?    0    :  ?  :    0 ; // Transition on non-clock signal. Use ? for clock.
  ?  f    ?    :  ?  :    - ; // No state transitions on inactive clock edge.
endtable
endprimitive
```

Because transitions to and from unknown values, as well as stable unknown values, must be accounted for, the table can grow quite large, and modeling a UDP that qualifies for optimization can be a daunting task. It is recommended that RTL flops be used for RTL regression simulation.

## Transparent Latches

In order to qualify for optimization, latches must be coded with the following restrictions:

- There can be three, or fewer, scalar wires in the sensitivity list.
- The `always` block must have a single `if` with no `else`.
- You must use non-blocking assignments with zero or a constant delay.
- The left-hand side of the assignments must be a scalar reg.

Example:

```
module DL (q, d, en, rst);
output q;
input d, en, rst;
reg q;

always @(en or rst or d)
  if (~en & rst)
    q <= #2 d;
```

```
always @ (rst)
  if (~rst)
    q <= #2 0;

endmodule
```

## Coding Styles to Avoid

Various coding styles detract from the overall simulation performance of any simulator. Many inefficiencies can be avoided by using common sense. For instance, if a DFF module has two outputs and one is the inverse of the other (`q` and `qn`), it is more efficient to have `qn` driven by the inverse of `q` than it is to have two separate flip-flop functions to drive each output individually.

This section focuses on coding styles that have a negative impact on the performance of NC-Verilog in less obvious ways.

### Avoid Using Blocking Procedural Assignments

Although blocking assignments (`=`) are slightly faster than non-blocking assignments (`<=`), only a small part of simulation time is spent performing assignments, and the disadvantages of using blocking assignments outweigh the small performance gain.

All of the examples shown in the section on recommended coding practices use non-blocking assignments for two reasons:

- Using regular blocking procedural assignments can cause event-ordering differences within a time slice, which can cause race conditions. Non-blocking assignments help to prevent race conditions.
- The rules for optimization when adding delays to modules with blocking assignments are much more complex than the rules for adding delays to modules with non-blocking assignments. For blocking assignments to be optimized by NC-Verilog:
  - All of the assignments to the output reg must have the same delay, and the delay must be specified on the right-hand side of the assignment (even when assigning constants to the reg).
  - All paths through the module must be of the same delay, even if the path does not eventually drive the reg.
  - If there is a conditional assignment to an output with a delay, and there is no assignment done for the non-active condition (an `if` with no `else`), there still must be a null statement with the same delay. The reasoning behind this is that the delay

in getting back to wait on the clock must be the same for all paths through the module.

For these two reasons, it is recommended that you use non-blocking assignments as much as possible.

### Use Regs Instead of Nets Wherever Possible

Verilog nets use more memory than regs. It also takes much longer to update nets than it takes to update regs.

Avoid complex continuous assignments.

#### **Bad:**

```
wire q = (en) ? d : q;
```

#### **Good:**

```
always @(posedge en or d)
    if (en) q <= d;
```

### Avoid always Blocks That Wait at Different Points

Procedural blocks that wait on different signals at different points in the block have a negative impact on performance. For example, a common coding style is to model a DFF to trigger only when the input changes, and then to assign the output at the next active edge of the clock, as shown in the following example:

```
module BAD_DFF (q, d, clk);
    output q;
    input d, clk;
    reg q;

    always @(d)
        @(posedge clk)
            q <= d;

endmodule
```

This is inefficient because it is dynamically changing the block's trigger, which adds overhead to the process of updating the effective fanout of the signal every time it changes. In addition, because of the multiple separate sensitivities in the block, a more complex data structure needs to be used in order to trigger it.

Because the NC-Verilog optimization algorithms avoid execution of `always` blocks when the non-clock inputs have not changed, the `always` block can be coded more efficiently as follows:

```
always @(posedge clk)
  q <= d;
```

It should also be noted that many synthesis tools do not accept code styles like that used in the `BAD_DFF` module shown above. In addition, this piece of code will not be optimized very well by a cycle simulator, if the cycle simulator even accepts it. So using the recommended coding style will enhance both NC-Verilog and cycle simulator performance, and will make the code more synthesizable.

### Avoid Code That Causes Zero-Delay Glitches

The following coding style is often used to ensure a default and to prevent inferred latches.

```
always @ (a or b or c)
begin
  {h, j, k} = 3'b0;
  case {a, b, c}
    3'b001 : k = 1'b1;
    3'b010 : j = 1'b1;
    3'b011 : k = 1'b1;
    3'b111 : h = 1'b1;
  endcase
end
```

The following style is more readable, prevents inferred latches, and has no unintended glitches.

```
always @ (a or b or c)
case {a, b, c}
  3'b001 : {h, j, k} <= 3'b001;
  3'b010 : {h, j, k} <= 3'b010;
  3'b011 : {h, j, k} <= 3'b001;
  3'b111 : {h, j, k} <= 3'b100;
  default : {h, j, k} <= 3'b000;
endcase
```

### Avoid Clocks with Skew

Some designs contain clock signals that transition from 0->X->1, and then from 1->X->0. Often, this all happens within the same time slice, with each transition representing a different delta cycle. Sometimes this behavior is intentional, such as to simulate clock skew, but most of the time it is not intentional.

This behavior on clock signals can cause problems because the designer is typically unaware of it, and thus the design may not be coded to handle it properly. Problems may arise in simulation if blocking procedural assignments are used (remember, non-blocking assignments do not assign the new value until the end of the time slice). Also, most UDP's are not constructed to handle this properly.

Clocks with skew also generate many unnecessary events. For example, an `always @(posedge clk)` block will run on the 0->X transition as well as on the X->1 transition. This means that every time the clock goes from 0->1, the `always` block executes twice. This obviously has a detrimental effect on simulation performance.

The most obvious way to detect this behavior is to insert a `$display` statement with a `$time` call into an `always` block that is triggered by the clock to see if it executes more than once in a particular time slice. You can also set an object breakpoint on the clock signal, and then continue the simulation a couple of times to see if it stops at the same simulation time more than once.

In general, intentionally modeling a clock to perform this behavior has more drawbacks than it is worth, for both performance and accuracy reasons. It's also good practice to make sure that clock signals in a design do not do this unintentionally.

### Avoid Assigning an Output reg to Itself

Assigning an output reg back to itself as part of a “hold state” disables the optimization algorithm. The algorithm requires that a constant or a wire (not a reg) be assigned to the reg. In the following example, the assignment `q <= q;` is not only unnecessary, but it also disables the optimization for this flop.

```
module BAD_DFF (q, d, clk, rst);
output q;
input d, clk, rst;
reg q;

always @(negedge rst)
  q <= 0;

always @(posedge clk)
  if (rst)
    q <= d;
  else
    q <= q;

endmodule
```

Modeling this flop in one of the recommended ways of modeling asynchronous reset will allow the algorithm to optimize this DFF, and it will still be synthesizable.

```
module DFF (q, d, clk, rst);
output q;
input d, clk, rst;
reg q;

always @(negedge rst)
  q <= 0;

always @(posedge clk)
  if (rst)
    q <= d;

endmodule
```

## Avoid Unnecessarily Wide Buses

Many designs require signals to be greater than 64 bits wide. Frequently, though, a design will have signals that are wider than necessary. Common examples of this are address registers and indices such as a for-loop counter. This is usually a result of the designer trying to be safe and not overflow the register.

These unnecessarily wide buses can cause severe performance slowdowns because the assembly code that gets generated can only operate using 32-bit registers, and thus must use more instructions for operations on signals larger than this.

A specific example of this is multiplication. If the left-hand side (the result) of a multiplication operation is wider than 64 bits, the simulator cannot use an inline multiply. In this case, NC-Verilog must use a function called `vm_mult`, and this will generate many more assembly instructions. To help detect this kind of problem, you can run the profiler. If the function `vm_mult` is showing up significantly in the profile, there might be multiplication operations writing to excessively large signals. See “[Using the Profiler to Identify and Eliminate Simulation Bottlenecks](#)” on page 697 for details on using the profiler.

A full understanding of the design specifications can help to avoid this problem. An index variable needs to be only wide enough to be able to handle the largest index accessible. For a for-loop counter, using the variable type `integer` (an integer is basically a 32-bit reg) should be more than enough. In the case of the address register, workstations that use 32-bit addresses cannot address enough memory to represent a Verilog memory that requires a 64-bit address. NC-Verilog produces a warning that it truncates the address register to 32 bits, but the point is that a lot of Verilog code contains unnecessarily wide buses which, in turn, cause an unnecessary reduction in simulation throughput.

## Avoid Unnecessary Vector Expansion

The simulator can simulate a change on one big object faster than it can simulate many changes on many small objects. A practical example would be a 32-bit bus. NC-Verilog can more easily update one 32-bit value than it can update 32 individual one-bit values. It is therefore more efficient to leave buses intact and not break them out into bits or parts, if possible. This will greatly reduce the number of values that the simulator must update and track.

There will, of course, be many design requirements where a bus has to be split out, but many designs do this unnecessarily. Netlisters are a common culprit when they netlist out all of the individual bits of an entire bus when they could just use the bus as a whole entity. Another example is declaring a bus with the `scalared` keyword, when it is never split out.

It is also important to note that driving a bit-select or a part-select of a wire forces the wire to be expanded. In addition, connecting a bit-select or part-select of a wire to a port will also force it to be expanded. Simply reading a bit or part from a wire will not force it to be expanded.

The following example illustrates this behavior. In this example, `test.a` will be expanded because a part-select of it is connected to a port. This wire will be expanded into 64 one-bit signals. On the other hand, `test.b` will be expanded into individual bits because one bit is broken out and driven individually.

```
module test();
wire [63:0] a;
wire [31:0] b;

submod U1 (a[31:0], b);

endmodule

module submod (in, out);
input [31:0] in;
output [31:0] out;
wire [31:0] out;

assign out[31] = ~in[31];
assign out[30:0] = in[30:0];

endmodule
```

The two `assign` statements in `submod` can be replaced by the following assignment to avoid vector expansion of `test.b`:

```
assign out = in ^ 32'h8000;
```

A common example of code that causes vector expansion is the following code fragment, which performs a sign extension operation:

```
wire [30:0] in;
wire [31:0] out;

assign out[31] = in[30];
assign out[30:0] = in;
```

This can be rewritten in the following way to prevent `out` from being expanded:

```
assign out = {in[30], in};
```

The following code fragment represents a more general-purpose method for sign extension, also without causing `out` to be expanded:

```
wire [Win:0] in;
wire [Wout:0] out;

assign out = { {(Wout-Win){in[Win]}}, in };
```

NC-Verilog recognizes this last piece of code as sign-extension and thus simulates it more efficiently.

## Avoid Excessive Hierarchy and Hierarchical References

Avoid excessive layering of library cells. This causes unnecessary extra connectivity, and adds to design size and complexity.

Also avoid the use of hierarchical paths to refer to objects (for example, `top.core.a`) wherever possible. Referencing an object from outside of its module causes widespread disabling of optimizations. Performing writes to an object from outside of a module has a more negative impact than reading its value from outside the module, but both will impact the performance of that module.

Besides having a negative impact on simulation performance, hierarchical references should be avoided for other reasons. They not only make the code more difficult to debug, but they also make it less flexible, in that you cannot as easily pull pieces out and plug in others, or reuse the testbench code somewhere else.

With the performance impact coupled with the effect they have on code reuse and readability, hierarchical references should be avoided as much as possible in both designs and testbenches.

## Refining the Testbench Strategy

This section discusses testbench strategies that can improve simulation performance.

### Use C and Tcl

Writing testbenches in Verilog can be inefficient because if you change the HDL you must recompile and then re-elaborate the entire design to generate a new snapshot. Consider alternatives to writing your testbenches in HDL, such as using C or C++.

Cadence recommends that you also look into its free open-source TestBuilder product. For information on TestBuilder go to:

<http://www.testbuilder.net/>

### Use \$readmemb or \$readmemh for Vectors

Testbench stimulus is often applied in one large `initial` block that executes for the length of the simulation, constantly delaying and doing more assignments to regs. There can be tens of thousands of these vectors in the `initial` block. This causes a compiled-code simulator, such as NC-Verilog, to generate a single, huge chunk of compiled code. Not only does this not simulate as well, but it can cause the code generator itself to take up enormous amounts of memory which, in turn, can cause swapping and thus severe slowdown.

In most cases, this stimulus code is machine-generated by a third-party tool over which you have no control. NC-Verilog contains some performance enhancements that speed up large stimulus `initial` blocks and that reduce memory overhead for both elaboration and simulation. However, if you create your own stimulus with this style, or if you develop a tool to create stimulus, it is better to create a stimulus file and to read this in with a `$readmemb` or `$readmemh` system task.

This methodology has several advantages over applying the stimulus in an `initial` block:

- It avoids the performance and memory capacity problems associated with huge `initial` blocks.
- It keeps the stimulus separate from the Verilog HDL code. Because of NC-Verilog's incremental compilation, it is more efficient to leave intact as much Verilog code as possible to minimize the amount of time spent compiling. Reading in a stimulus file containing vectors with a `$readmemb` or `$readmemh` lets you change the data file without rebuilding the design. This can provide tremendous efficiency gains, especially if the stimulus changes often and is large.

## NC-Verilog Simulator Help

### Maximizing Simulation Performance

---

- It lets you separate the clock that drives the test from the clock that drives the design, which eliminates race conditions. For example:

```
reg[127:0] stim_mem[0:20000];
assign my_inputs = stim_mem[i];

initial
begin
    $readmemh("stim0.dat", stim_mem);
    i = 0;
    clk = 1'b0;
    testclk = 1'b0;
end

always @(testclk)
begin
    i <= i + 1;
    #1;
    clk <= !clk;
    testclk <= #9 !testclk;
end

always @(i) if (i >= 20000) $finish(2);
```

The following example shows how you can apply stimulus at irregular time intervals with `$readmemh`.

```
module top;
// input buses
reg [15:0] in1;
reg [15:0] in2;
reg [7:0] in3;
// delay value
reg [15:0] delay;
// counter of vector number
integer vecnum;
// table of input vectors
// width = sum of input widths + delay width
// depth = maximum number of vectors expected
reg [71:0] stimtable[1023:0];

// top level design module
my_design DUT(in1, in2, in3, out);
```

```
initial
begin

    // read in vectors from file
    $readmemh("vectors.dat", stimtable);
    // start with first vector
    vecnum = 0;

    // loop through vectors
    forever
        begin
            // assign input values and delay before next vector
            {in1, in2, in3, delay} = stimtable[vecnum];
            if (delay == 0)// use 0 delay to indicate end of vectors
                $finish(2);
            else // else wait for delay time
                #(delay);
            // next vector
            vecnum = vecnum + 1;
        end
    end

endmodule
```

Here is the example stimulus file, vectors.dat. The extra spaces, underscores, and comments in this file will slow down \$readmem initialization slightly. They are added here to make the example clearer.

```
// in1_in2_in3_delay// time 0
0000_0000_00_0014// time 20
0001_0001_ff_000a// time 30
0002_0002_ff_000a// time 40
0004_0003_ff_000a// time 50
0008_0004_ff_000a// time 60
0010_0005_00_000a// time 70
0020_0006_ff_000a// time 80
0040_0007_ff_000a// time 90
0080_0008_ff_000a// time 100
0100_0009_ff_0014// time 120
0200_000a_ff_000a// time 130
0400_000b_ff_000a// time 140
0800_000c_ff_000a// time 150
```

```
0000_000d_ff_000b// time 161  
0000_0000_00_0000// end of vectors
```

**Note:** If no addressing information is specified in the system task, and if no address specification appears in the data file, the default start address for loading the memory for NC-Verilog is the lowest address given in the declaration of the memory. Data is read into the memory from the lowest address to the upper address. This matches the behavior of Verilog-XL, but differs from the IEEE standard, which specifies that the default start address for loading the memory is the left-hand address given in the declaration.

For example, given the following memory declaration, NC-Verilog will load the memory, starting with the lowest address (in this case, the address on the right).

```
reg[7:0] mem{256:1};
```

To match the behavior specified in the standard, you can declare the memory with the lowest address on the left, or you can specify both the starting and ending address to \$readmemb or \$readmemh.

## Use \$test\$plusargs for Conditional Code

A common coding style is to use `ifdef compiler directives in a Verilog module to control the execution of certain statements when the `define compiler directive appears in the source code, or when the -define (ncverilog +define) compile-time option appears on the command line. For example, you could control the dumping of values to a VCD file with the following code:

```
initial  
begin  
`ifdef dumpon  
$dumpfile("results.vcd");  
$dumpvars;  
`endif  
end
```

You can then dump values to a VCD file by compiling with the following command:

```
% ncverilog -define dumpon test.v  
(% ncverilog +define+dumpon test.v)
```

This method of coding conditional code requires recompilation every time that the compile-time option changes.

A more efficient method is to use the \$test\$plusargs system function to check for the presence of a plusarg run-time option. For example, the code shown above can be written as follows:

```
initial
  if ($test$plusargs("dumpon"))
begin
  $dumpfile("results.vcd");
  $dumpvars;
end
```

In this example, you can enable dumping by including the `+dumpon` option on the `ncsim` command line. No recompilation is necessary.

```
% ncsim +dumpon snapshot_name
```

The following example illustrates how to use `$test$plusargs` to control which stimulus file is used for a particular simulation.

```
initial
  if ($test$plusargs("test01"))
  $readmemh("test01.dat", stim_mem);

  if ($test$plusargs("test02"))
  $readmemh("test02.dat", stim_mem);

  if ($test$plusargs("test03"))
  $readmemh("test03.dat", stim_mem);

  if ($test$plusargs("test04"))
  $readmemh("test04.dat", stim_mem);
```

By using the `$test$plusargs` system function, you can change the stimulus without rebuilding the design. This method is also an easy way to run multiple simulations in parallel. For example:

```
% ncsim snapshot_name +test01 &
% ncsim snapshot_name +test02 &
% ncsim snapshot_name +test03 &
% ncsim snapshot_name +test04 &
```

## Create Self-Checking Tests

As mentioned in a previous section, one of the biggest bottlenecks in designs and testbenches is excessive I/O: too much file writing, too much message displaying, or too much waveform dumping. True regression runs are self-checking and should require only minimal screen and file I/O. Create tests that print information only if an error is encountered or upon successful completion of the test. Should one of the tests find a bug and cause a particular test to fail, that test can be rerun with waveforms and informative messages. You

should also turn off unnecessary warning messages, and include code to exit the simulator if the error count is high.

In the following code fragment, messages are printed only if there is an error, and the simulation stops if there are more than 20 errors.

```
always @(i)
begin
#(`HCYCLE - 1);
if (response !== resp_mem(i))
begin
$display("Error at vector %d", i);
err_cnt = err_cnt + 1;
end
end

always @(err_cnt)
if (err_cnt >= 20)
begin
$display("Too many errors. Exiting");
$finish(2);
end
```

## Avoiding Unnecessary Recompilation

This section contains information on a few simple steps that you can take to minimize the amount of recompilation that is necessary when changes are made to modules.

### Run the Parser in Update Mode (*ncvlog -update*)

If you are running NC-Verilog using single-step invocation (*ncverilog*), this is the default.

If you are running NC-Verilog in multi-step invocation, there are three ways to update a model:

- Running *ncsim* with the *-update* option.
- Running the *ncupdate* utility, followed by *ncsim*.
- Running *ncvlog* with the *-update* option, followed by *ncelab* and *ncsim*.

The first two methods can be efficient ways to provide a quick design change turnaround when you have edited a design unit, but they have restrictions, and, depending on the kind of change that you have made, may not update correctly. In particular, the changes cannot:

- Modify the hierarchy (for example, by instantiating an additional module or a different module).
- Modify parameterization (for example, by instantiating DFF\_inst109 as 8-bits wide instead of 4-bits).
- Add a source file.
- Modify names of source files to be used in the build (for example, by changing `include files).
- Change a design unit in a way that introduces a new cross-file dependency.

Because of these restrictions, using `ncvlog -update` is recommended. You can add this option to the definition of the `NCVLOGOPTS` variable in the `hdl.var` file so that it is always used.

## Eliminate Cross-File Inheritance

If you place compiler directives and similar information in one module and assume they will be active for another module, you always need to compile the modules in a specific order. Moreover, more modules than necessary are recompiled when you use the simulator's update feature. When you revise a module that has a cross-file inheritance link, every module that depends on the inheritance must be recompiled to insure that each is up-to-date. This will substantially reduce the efficiency of your design revision cycle.

For example, consider the following situation where `source2.v` has a cross-file dependency on `source1.v`. In this example, the `'timescale` directive also applies to `mod_2`. If `mod_1` is revised, both modules must be recompiled.

```
// File source1.v                                // File source2.v
'timescale 1ns/1ps                                module mod_2;
module mod_1;                                       output o;
reg in;                                            input e, i;
wire out, a, b;                                     wire t, t1;
...                                                 assign t1 = t;
...                                                 ...
...                                                 endmodule
endmodule
```

In addition, cross-file inheritance may spawn across the logical libraries. As a result, the way source code is compiled in one library can affect another logical library.

To avoid this situation, put all global information (such as `timescale, `define, and so on) in a file that can be included by the source files. If you use this approach, no information is inherited by one module from another.

As in Verilog-XL, the `resetall compiler directive does not reset macros that have been set with `define. You can use the `undefineall compiler directive before `resetall to reset macros.

**Note:** The `undefineall directive is not currently supported by Verilog-XL, so you must surround it with `ifdef INCA. Cells that started with `resetall should begin with:

```
'ifdef INCA  
'undefineall  
'endif  
'resetall
```

## Use One Module per File

If you have multiple modules in a design file, you may recompile modules that do not have to be recompiled during incremental compile. Using one module per file is more efficient, especially for source files that are changing frequently.

However, putting multiple modules in a static or infrequently changed library file is desirable and even preferable.

## Avoid Modules in `include Files

If you put modules in a `include file, every module that references it is dependent on that module file. Whenever a module changes, the module that includes it must also be recompiled. The best use of `include files is for shared control information, such as `define, `timescale, and so on.

## Avoid Compile-Time Conditional Code

If possible, use the \$test\$plusargs system function to check for the presence of a simulation-time plusarg option instead of using `ifdef compiler directives and the compile-time -define (or +define for ncverilog) command-line option.

See “[Use \\$test\\$plusargs for Conditional Code](#)” on page 692 for more information.

## Using the Profiler to Identify and Eliminate Simulation Bottlenecks

The profiler is a tool that measures where CPU time is spent during simulation. Although it was developed primarily to help Cadence R&D diagnose performance bottlenecks in the simulator, some of the information in the output file can help you to identify inefficient HDL coding practices. Once you have determined what code the simulator is spending most of its time running, improving the efficiency of this code will have the greatest effect on simulation performance.

The profiler works by interrupting the simulation at regular intervals (currently 100 times per second) and noting what was executing at that time. It keeps track of the number of "hits" on different activities, which approximates the amount of CPU time spent in these activities.

The profiler is easy to run and has minimal impact on simulation performance and memory usage. To run the profiler, use the `-profile` command-line option when you invoke the simulator (`ncsim`). Use `+ncprofile` if you are running in single-step invocation mode with the `ncverilog` command.

```
% ncsim -profile snapshot_name  
% ncverilog -f arguments_file +ncprofile
```

When the simulator exits, the profiler creates a file called `ncprof.out` in the run directory.

Each profile begins with a header that provides general information. This includes the version of `ncsim`, the operating system and version, and a description of the computer hardware on which the simulation was executed. The header also includes the number of interrupts per second, and the same total memory and CPU usage information that the `ncsim -status` option provides.

The information contained in the header can reveal performance problems such as insufficient physical memory for the size of the simulation, or low CPU utilization due to a busy machine or waiting for I/O.

The following is an example of the header section in a profile.

```
ncsim: v03.30.(s001): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.  
  
SunOS cs-sj1-238 5.7 Generic_106541-11 sun4u sparc Sun_Microsystems  
SUNW,Ultra-80, 1 CPU, 450 MHz, 4096 Meg RAM, 100 hits/sec, (larrybird)  
  
Memory Usage - 14.2M program + 446.6M data + 6.0M profile = 466.8M total  
CPU Usage - 9.5s system + 1087.1s user = 1096.6s total (98.2% cpu)
```

After the header, the profile is divided into three sections: *Stream Counts*, *Most Active Modules*, and *Stream Type Summary Counts*. These sections summarize the profile information in different ways, but have the same basic structure. There are four columns of information:

- The first column shows the percentage of the total hits that were detected in a given activity.
- The second column displays the raw number of hits that were detected in a given activity.

The percentage shown in the first column is simply the raw number of hits as a percentage of the total given at the top of the section.

Multiplying the raw number of hits by the interrupt rate indicated in the header, gives the approximate CPU time spent on that particular activity.
- The third column shows the number of instances, when applicable.
- The fourth column gives the name or description of the activity.

Within each section, the activities are sorted in descending order of number of hits. That is, the most time-consuming activities appear at the top of the list.

The following sections describe the three sections of the profile in more detail.

## Stream Counts

The section titled *Stream Counts* provides detailed information about time spent in individual generated code streams. These code streams correspond to specific HDL source constructs. They include:

- always blocks
- initial blocks
- continuous assignments
- tasks and functions
- non-blocking assignments
- quasi-continuous assignments
- parallel block sub-processes (statements inside fork/join)
- Logic primitives for gates and UDPs (Because logic primitives share the same code between different instances, the profiler cannot indicate which gate instances are taking up the most time.)

There are also separate streams for certain kinds of complex statements that have to operate asynchronously from the blocks containing them. Anonymous (or implicit) continuous assignments are inserted when regs or expressions are attached to module ports. Timing output delay elements are inserted to implement path delays.

There are other activities that are not clearly related to any particular source construct or that do not use generated code. For example, there are run-time library functions that implement system tasks or complex arithmetic operations. These can often be recognized by their names. The run-time library function `rtl_readmem`, for example, implements the `$readmemb/$readmemh` system tasks. There are categories for VPI/PLI and tracing support. There are also streams called “methods”, which perform internal simulator operations, usually related to propagating fanout. Some of these will print a useful description of their purpose, but most of them require extensive knowledge of the simulator to interpret.

The beginning of the *Stream Counts* section is the first place to look for inefficiency. If a few streams are taking up most of the simulation time, simulation cannot be sped up significantly without reducing the time in those streams. If an HDL construct appears unexpectedly high in the list, it may be written very inefficiently. Perhaps it does not qualify for an optimization that applies to other similar code.

The guidelines given earlier in the chapter may help in recognizing the inefficiencies after the profiler has identified where they are. It may also be worthwhile to experiment with different ways of writing something to see what effect it has.

Large amounts of time spent in methods tends to indicate a design with a low level of abstraction. In particular, methods with `BYTE` in the name update the values of one-bit nets. If these methods show up high in the profile, the simulator is spending a lot of time updating scalar nets or individual bits of expanded vector nets. In a gate-level design, this is part of the price for simulating at a low level of abstraction. In a behavioral design, this may mean that vector nets have been forced to expand. In addition to the time spent in the methods, expansion slows down the streams that drive or read the vector net.

In some cases, the simulator may optimize HDL constructs by grouping several of them together in a single stream. The profiler will report all of the time against one of them. For example, several consecutive non-blocking assignments may be grouped together, with all of the time reported against the first one. This can look like the first one is less efficient than the others, but changing the order will move the time to a different one. This kind of optimization makes it harder to recognize actual inefficiency.

Note that streams with counts below a certain threshold (currently set to 0.1% of the total hits), are not listed in the stream section to keep the list from getting too long. While individual streams with counts below the threshold do not affect the performance much, their combined effect may have a significant impact.

## NC-Verilog Simulator Help

### Maximizing Simulation Performance

---

The following example shows the *Stream Counts* section from a profile.

```
-----
Stream Counts (259 hits total)
-----
%hits #hits #inst name
 54.4 141 [ ] User-defined primitive 'UDP_DFF' input 'cp' (zero delay) (method)
   6.9   18 [ ] User-defined primitive 'UDP_LATCH_NF' input 'cp' (zero delay)
(method)
   6.6   17 [ ] Method SSS_MT_DU_BYTENFW (method)
   6.2   16 [ ] Logic primitive 'and' (zero delay) (method)
   6.2   16 [ ] User-defined primitive 'UDP_LATCH' input 'cp' (zero delay)
(method)
   3.9   10 [ ] User-defined primitive 'UDP_DFF' input 'cp' (zero delay) (method)
   3.1    8 [ ] User-defined primitive 'UDP_DFF' input 'cp' (zero delay) (method)
   2.7    7 [ ] User-defined primitive 'UDP_DFF' input 'cp' (zero delay) (method)
   2.7    7 [ ] Logic primitive 'not' (1 outputs) (zero delay) (method)
   1.2    3 [ ] User-defined primitive 'UDP_DFF' input 'cp' (zero delay) (method)
   1.2    3 [ ] User-defined primitive 'UDP_DFF' input 'cp' (zero delay) (method)
   0.8    2 [ 1] Always stmt (file: ./addsub_top.v, line: 187 in worklib.top
[module])
   0.8    2 [ ] outside engine
   0.4    1 [ ] Method SSS_MT_POSEDGEN (method)
   0.4    1 [ ] Logic primitive 'or' (zero delay) (method)
   0.4    1 [ ] Logic primitive 'nor' (zero delay) (method)
   0.4    1 [ ] Method SSS_MT_RETURN_BYTE (method)
   0.4    1 [ ] Method SSS_KM_FINDRFT (method)
   0.4    1 [ 1] Always stmt (file: ./addsub_top.v, line: 259 in worklib.top
[module])
   0.4    1 [ 1] Always stmt (file: ./prog_ram_syn.v, line: 41 in
worklib.prog_ram_syn [module])
   0.4    1 [ 1] Always stmt (file: ./prog_ram_syn.v, line: 43 in
worklib.prog_ram_syn [module])
   0.4    1 [ 4] Continuous Assignment (file: ./addsub_top.v, line: 410 in
worklib.clk_gate_ran_r [module])
```

The example *Stream Counts* section shown above contains a category called “Outside engine.” This category, and one other called “Engine support”, are catch-all categories for activities that cannot be otherwise categorized. These categories can also appear in the *Stream Type Summary Counts* section.

- “Outside engine” is a catch-all description for external processing. This refers to C code that is executed outside of the main simulation engine. This could be caused by the GUI or by intensive TCL processing. A high number for “Outside engine” could also be the

result of having a complex simulation environment in which foreign models and other products are being used in conjunction with the simulator.

- “Engine support” refers to time spent in support functions that are called from within the simulation engine and for time spent while the model is running that is not otherwise categorized. This could be user PLI code or some other uncategorized function.

## Most Active Modules

The second section of the profile, *Most Active Modules*, summarizes the stream counts by module. For each module listed, the sum of the counts in all of the streams in that module is given. This can be useful when a module is taking a lot of time due to the combined time in a lot of streams that are below the threshold for being listed in the *Stream Counts* section. It may also provide useful information to someone who is familiar with the design at the module level.

Counts that are not clearly associated with a particular module are omitted from this section.

The following is an example of the *Most Active Modules* section.

---

---

```
Most Active Modules (behavioral)
-----
%hits #hits #inst name
27.3    75 [     8] worklib.SDRAM:v (file: ../lib/sdram.v line: 215)
10.9    30 [     1] worklib.ddr_ctrl:v (file: ../rtl/ddr_ctrl.v line: 43)
 6.5    18 [     1] worklib.page_table:v (file: ../rtl/page_table.v line: 59)
 4.4    12 [     1] worklib.ddr_dxfr:v (file: ../rtl/ddr_dxfr.v line: 56)
 4.4    12 [     1] worklib.system:v (file: ./system.v line: 53)
 4.0    11 [     4] worklib.BusMaster:v (file: ../lib/bm.v line: 169)
 4.0    11 [     1] worklib.gg_test:v (file: ./gg_test.v line: 49)
 3.3     9 [     1] worklib.arbiter:v (file: ../rtl/arbiter.v line: 51)
 2.5     7 [     3] worklib.sync_fifo:v (file: ../rtl/sync_fifo.v line: 30)
 1.8     5 [     1] worklib.agp_test:v (file: ./agp_test.v line: 26)
 1.8     5 [     1] worklib.data_cyc:v (file: ../rtl/data_cyc.v line: 46)
 1.5     4 [     1] worklib.add_fifo:v (file: ../rtl/add_fifo.v line: 51)
 1.5     4 [     1] worklib.req_fifo:v (file: ../rtl/req_fifo.v line: 27)
 0.7     2 [     3] worklib.bank_sel:v (file: ../rtl/bank_sel.v line: 35)
 0.4     1 [     1] worklib.ddr_top:v (file: ../rtl/ddr_top.v line: 63)
 0.4     1 [     1] worklib.max_test:v (file: ./max_test.v line: 23)
 0.4     1 [     8] worklib.mem_bank:v (file: ../lib/mem_bank.v line: 1)
 0.4     1 [     1] worklib.mem_req:v (file: ../rtl/mem_req.v line: 56)
```

## Stream Type Summary Counts

The *Stream Type Summary Counts* section summarizes the stream counts by the type of stream or other activity. For example, there might be a total for logic primitives, timing checks, always or initial statements, non-blocking assignments, continuous assignments, and so on. As with the *Most Active Modules* section, this section includes streams with counts too low to appear separately in the first section. This section gives a general idea of where simulation time is being spent.

The summary makes it easier to identify widespread inefficiencies in the simulation. For example, large amounts of time spent on probing, file I/O, and PLI will show up most clearly in this section. Time spent directly on timing features is also summed up here. If they are taking a lot of time and are not actually needed, these timing features can be turned off with command-line options. A design that is supposedly behavioral may actually be spending most of its time simulating gate-level portions. Replacing those portions with equivalent higher-level models could improve performance dramatically.

Here is an example *Stream Type Summary Counts* section:

```
-----  
Stream Type Summary Counts (275 hits total)  
-----  
%hits #hits #inst name  
30.2    83 [ 1456] Timing checks  
25.5    70 [  229] Always statements  
17.5    48 [      ] Standard methods (mostly fanout propagation)  
11.3    31 [  759] Non-blocking assignments  
 8.4    23 [  324] Continuous assignments  
 4.7    13 [   88] Initial statements  
 3.6    10 [   41] Anonymous continuous assignments  
 2.9     8 [ 193] Verilog tasks  
 2.5     7 [   26] Verilog functions  
 2.5     7 [      ] Wire evaluation  
 2.2     6 [      ] System tasks/functions or library functions  
 1.5     4 [      ] Outside engine  
 0.7     2 [   80] Parallel block sub-processes
```

---

## Timing Checks

---

This chapter contains the following sections:

- [Overview](#)
- [Timing Check System Tasks](#)
- [Using Edge-Control Specifiers](#)
- [Using Notifiers](#)
- [Enabling Timing Checks with Conditioned Events](#)
- [Negative Timing Check Limits in \\$setuphold and \\$recrm](#)
- [Timing Violation Messages](#)
- [SDF Annotation of Timing Checks](#)

## Overview

A timing check verifies the timing performance of a design by making sure that critical events occur within given time limits. A timing check performs the following steps:

1. Determines the elapsed time between two events.
2. Compares the elapsed time to a specified minimum or maximum time limit.
3. Reports a timing violation if the elapsed time occurs outside the specified time window.

Timing check information can be included by using the timing check system tasks available in the Verilog HDL or by using SDF annotation. This topic deals primarily with using the system tasks. See “[SDF Annotation of Timing Checks](#)” on page 745 for details on SDF annotation.

The timing check system tasks are invoked in specify blocks. They can contain:

- Edge-control specifiers, which control events in timing checks based on specific edge transitions between 0, 1, and x. (See “[Using Edge-Control Specifiers](#)” on page 722 for details.)
- Notifiers, which specify user-defined responses to timing violations. (See “[Using Notifiers](#)” on page 723 for details.)
- Conditioned events, which tie the occurrence of timing checks to the value of a conditioning signal. (See “[Enabling Timing Checks with Conditioned Events](#)” on page 726 for details.)

(See “[Specify Blocks](#)” on page 758 for details on using these statements.)

You can invoke the following timing check system tasks:

- [\\$setup](#)
- [\\$hold](#)
- [\\$setuphold](#)
- [\\$width](#)
- [\\$period](#)
- [\\$skew](#)
- [\\$recovery](#)
- [\\$removal](#)

- \$recmem
- \$nochage

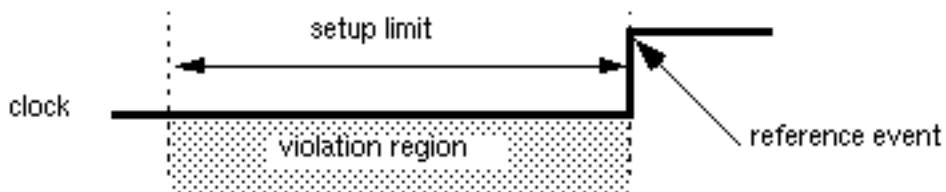
**Note:** If you do not want to execute timing checks, you can improve processing performance by disabling timing checks by elaborating with the `-notimingchecks` option. For example,

```
% ncelab -notimingchecks top
```

## Timing Check System Tasks

### \$setup

The `$setup` system task determines whether a data signal remains stable long enough before a transition in a control signal, such as a clock signal that latches data in memory. A violation occurs when a change to the data signal occurs within a specified time limit before the transition at the control signal. The following figure illustrates the violation region specified by the `$setup` system task.



The `$setup` system task has the following format:

```
$setup(data_event, reference_event, setup_limit [,notifier]);
```

The `$setup` system task arguments are as follows:

---

<b>\$setup Arguments</b>	<b>Description</b>
<code>data_event</code>	Transition at a data signal that initiates the timing check. This is a module input or inout that is a scalar or vector net.
<code>reference_event</code>	Transition at a control signal that establishes the reference time for tracking timing violations on the data event. This is a module input or inout that is a scalar or vector net.

\$setup Arguments	Description
<i>setup_limit</i>	Positive constant expression or specparam that specifies the minimum interval between the data event and the reference event. Any change to the data signal within this interval results in a timing violation.
<i>notifier</i> (optional)	Register whose value is updated whenever a timing violation occurs. You can use notifiers to define responses to timing violations. (See “ <a href="#">Using Notifiers</a> ” on page 723 for details.)

**Note:** If the reference event and the data event occur simultaneously, \$setup performs the timing check before it records the new data event value. Therefore, no violation is reported.

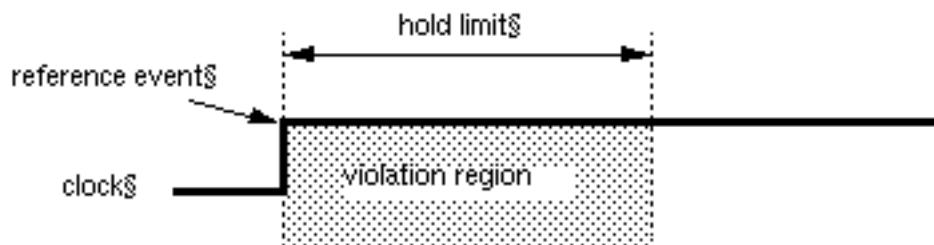
Example:

In the following example, the \$setup system task sets up a time window 10 time units before a positive transition on `clock`. A timing violation is reported if data transitions within this interval.

```
specify
  specparam setup_param=10;
  $setup( data, posedge clock, setup_param );
endspecify
```

## \$hold

The \$hold system task determines whether a data signal remains stable long enough after a transition in a control signal, such as a clock signal that latches data in a memory. The following figure illustrates the violation region specified by the \$hold system task.



The \$hold system task has the following format:

```
$hold(reference_event, data_event, hold_limit [,notifier]);
```

The \$hold system task arguments are as follows:

\$hold Arguments	Description
<i>reference_event</i>	Module input or inout transition at a control signal that establishes the reference time.
<i>data_event</i>	Module input or inout transition at a data signal that initiates a timing check against the value in <i>hold_limit</i> .
<i>hold_limit</i>	Positive constant expression or specparam that specifies the interval between the reference event and data event. Any change to the data signal within this interval results in a timing violation. If <i>hold_limit</i> is 0, a timing check does not occur.
<i>notifier</i> (optional)	Register whose value is updated whenever a timing violation occurs. You can use notifiers to define responses to timing violations. (See “ <a href="#">Using Notifiers</a> ” on page 723 for details.)

**Note:** If the reference event and the data event occur at the same time, the \$hold timing check records the new reference event time before it performs the timing check and a violation is reported.

Example:

In the following example, \$hold reports a violation if the time that elapses from the reference event (`posedge clk`) to a change in `data` is smaller than `hold_param(11)`. The notifier, `flag`, detects the timing violation behaviorally and performs some user-defined action.

```
specify
    specparam hold_param=11;
    $hold( posedge clk, data, hold_param, flag );
endspecify
```

## \$setuphold

The \$setuphold task combines the functionality of \$setup and \$hold into one system task. It also offers additional functionality in that you can specify a negative time specification for the setup limit or for the hold limit.

The \$setuphold system task has the following format:

```
$setuphold( reference_event, data_event, setup_limit, hold_limit
            [,notifier] [,tstamp_cond] [,tcheck_cond]
            [,delayed_reference] [,delayed_data] );
```

## NC-Verilog Simulator Help

### Timing Checks

---

The `$setuphold` system task arguments are as follows:

<b>\$setuphold Arguments</b>	<b>Description</b>
<code>reference_event</code>	Transition at a control signal that establishes the reference time for tracking timing violations on the data event. This is a module input or inout that is a scalar or vector net. The <code>reference_event</code> argument represents the lower bound event for <code>\$hold</code> and the upper bound event for <code>\$setup</code> .
<code>data_event</code>	Transition at a data signal that initiates the timing check. This is a module input or inout that is a scalar or vector net. The <code>data_event</code> argument represents the upper bound event for <code>\$hold</code> and the lower bound event for <code>\$setup</code> .
<code>setup_limit</code>	Constant expression or specparam that specifies the minimum interval between the data event and the reference event. Any change to the data signal within this interval results in a timing violation.  You can specify negative times for either the <code>setup_limit</code> or <code>hold_limit</code> arguments. The sum of the two arguments must be 0 or greater.
<code>hold_limit</code>	Constant expression or specparam that specifies the interval between the reference event and data event. Any change to the data signal within this interval results in a timing violation.  You can specify negative times for either the <code>setup_limit</code> or <code>hold_limit</code> arguments. The sum of the two arguments must be 0 or greater.
<code>notifier</code> (optional)	Register whose value is updated whenever a timing violation occurs. You can use notifiers to define responses to timing violations. (See “ <a href="#">Using Notifiers</a> ” on page 723 for details.)
<code>tstamp_cond</code> (optional)	Places a condition on the stamp event. For the setup check of <code>\$setuphold</code> , this argument places a condition on the transition of the data signal. For the hold check of <code>\$setuphold</code> , this argument places a condition on the transition of the reference signal.
<code>tcheck_cond</code> (optional)	Places a condition on the check event. For the setup check of <code>\$setuphold</code> , this argument places a condition on the transition of the reference signal. For the hold check of <code>\$setuphold</code> , this argument places a condition on the transition of the data signal.

<b>\$setuphold Arguments</b>	<b>Description</b>
<i>delayed_reference</i> (optional)	A delayed version of the reference signal generated by a negative timing check. See “ <a href="#">Negative Timing Check Limits in \$setuphold and \$recrem</a> ” on page 728 for more information.
<i>delayed_data</i> (optional)	A delayed version of the data signal generated by a negative timing check. See “ <a href="#">Negative Timing Check Limits in \$setuphold and \$recrem</a> ” on page 728 for more information.

Absent optional parameters must be indicated as null parameters by using commas. Do not add one or more commas after the last argument. For example, the following `$setuphold` task includes a *tcheck\_cond* argument:

```
$setuphold(posedge clk, data, 2, 3, , tcheck_cond);
```

**Note:** You cannot condition a `$setuphold` timing check by using the *tstamp\_cond* or *tcheck\_cond* arguments and a conditioned event. If you attempt to use both methods, only the parameters in the *tstamp\_cond* and *tcheck\_cond* arguments are effective, and a warning message is generated.

#### Example 1:

The following example shows how the `$setuphold` system task combines the functionality of `$setup` and `$hold`. The following invocation:

```
$setuphold(posedge clk, data, 3, 2);
```

is equivalent to the following two system tasks:

```
$setup( data, posedge clk, 3 );
$hold( posedge clk, data, 2 );
```

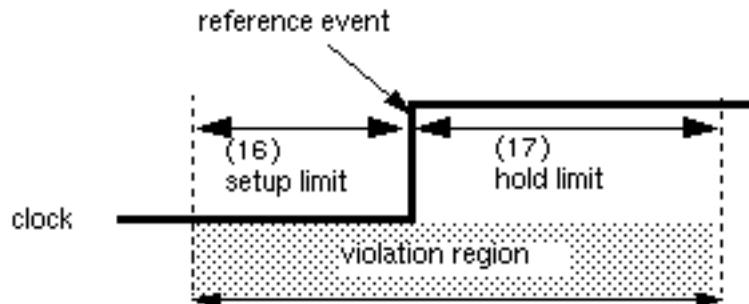
#### Example 2:

The following example illustrates `$setuphold` with positive time specifications for both *setup\_limit* and *hold\_limit* arguments.

```
specify
  specparam tSU=16, tHLD=17;
  $setuphold( posedge clk, data, tSU, tHLD );
endspecify
```

In this example, `$setuphold` reports a violation if the interval from a transition on the data signal to the positive edge of `clk` is less than `tSU` (16), enacting its `$setup` component. It also reports a violation if the interval from the positive edge of `clk` to a transition on the data signal is less than `tHLD` (17), enacting its `$hold` component.

The following figure shows the violation region:



Example 3:

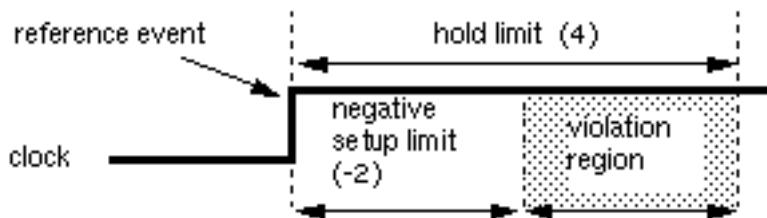
You can specify negative times for either the *setup\_limit* or *hold\_limit* arguments. The sum of the two arguments must be 0 or greater.

- A negative *setup\_limit* value specifies a period following a change in the reference signal.
- A negative *hold\_limit* value specifies a period preceding a change in the reference signal.

**Note:** Beginning with the LDV 3.3 release, the use of negative time specifications is enabled by default. With LDV 3.2 or a previous release, you must use the *-neg\_tchk* option on the command line when you invoke the elaborator if you use negative time specifications. If you do not specify this option, negative limits are set to 0 in the description or annotation, and a warning is issued.

The following figure illustrates the violation region for the following `$setuphold`, which specifies a negative setup limit:

```
$setuphold( posedge clk, data, -2, 4);
```

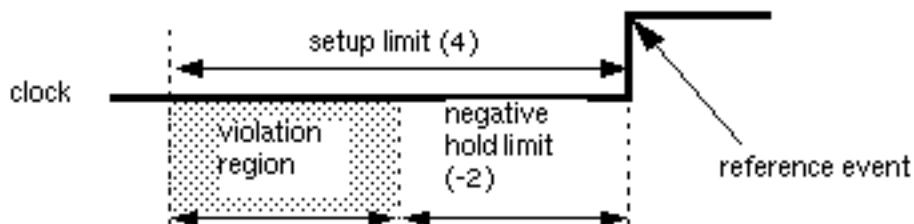


The following figure illustrates the violation region when you specify a negative hold limit.

## NC-Verilog Simulator Help

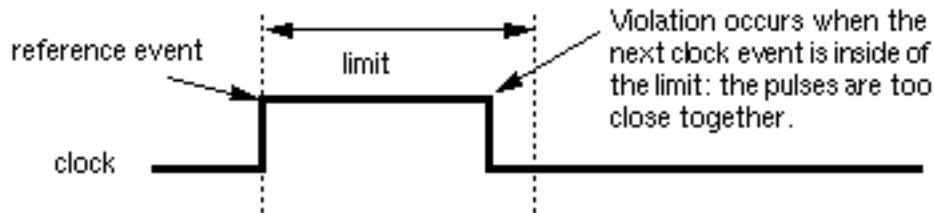
### Timing Checks

```
$setuphold( posedge clk, data, 4, -2 );
```



## \$width

The `$width` system task specifies the duration of signal levels from one edge transition to the opposite edge transition. A violation occurs when the pulse width is smaller than the specified limit. The following figure illustrates how a violation occurs with the `$width` system task.



The syntax for the `$width` system task is as follows:

```
$width(reference_event, limit [,threshold, notifier]);
```

Notice that no `data_event` argument is passed to `$width`. The data event for `$width` is derived from the reference event and is the reference event signal with the opposite edge.

## NC-Verilog Simulator Help

### Timing Checks

---

The `$width` system task arguments are as follows:

<b>\$width Arguments</b>	<b>Description</b>
<code>reference_event</code>	Transition at a control signal that establishes the reference time for tracking timing violations on the data event. This argument must be an edge-triggered event. A compilation error occurs if the reference event is not an edge specification.
<code>limit</code>	Positive constant expression or specparam that specifies the minimum interval between the time of the reference event and the time of the implicit data event.
<code>threshold</code> (optional)	Positive constant expression or specparam that specifies the largest ignored pulse width. This argument is used for timing analysis by Veritime. NC-Verilog ignores <code>threshold</code> , but compiles system calls to <code>\$width</code> that contain this argument.
<code>notifier</code> (optional)	Register whose value is updated whenever a timing violation occurs. You can use notifiers to define responses to timing violations. (See “ <a href="#">Using Notifiers</a> ” on page 723 for details.)

---

If you specify more than one edge control specifier, the edges must be either all rising (any of 01, 0X, X1) or all falling (any of 10, 1X, X0).

Example 1:

The following `$width` system task reports a violation if the interval from the reference event (`negedge clr`) to the implicit data event (`posedge clr`) is less than the limit specified by `width_param(12)`. Note that the data event and the reference event never occur simultaneously because they are triggered by opposite transitions.

```
specify
  specparam width_param=12;
  $width( negedge clr, width_param );
endspecify
```

Example 2:

You cannot use null arguments for \$width. If you pass a *notifier* argument, you must also supply the *threshold* argument. However, you do not have to specify either of these arguments. The following example shows legal and illegal calls.

// Legal calls

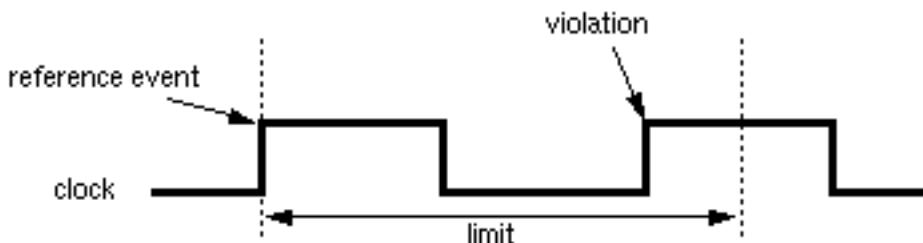
```
$width( negedge clr, lim );
$width( negedge clr, lim, thresh, notif );
$width( negedge clr, lim, 0, notif );
```

// Illegal calls

```
$width( negedge clr, lim, , notif );
$width( negedge clr, lim, notif );
```

## \$period

The \$period system task specifies the duration of signal levels from one edge transition to the same edge transition. A violation occurs when the pulse width is smaller than the specified limit. The following figure illustrates how a violation occurs with the \$period system task.



The \$period system task has the following format:

```
$period(reference_event, limit [,notifier]);
```

Notice that no *data\_event* argument is passed to \$period. The data event for \$period is derived from the reference event and is the reference event signal with the same edge.

## NC-Verilog Simulator Help

### Timing Checks

---

The \$period system task arguments are as follows:

\$period Arguments	Description
<i>reference_event</i>	Transition at a control signal that establishes the reference time for tracking timing violations on the data event. This argument must be an edge-triggered event. A compilation error will occur if the <i>reference_event</i> is not an edge specification.
<i>limit</i>	Positive constant expression or specparam that specifies the minimum interval between the time of the reference event and the time of the implicit data event.
<i>notifier</i> (optional)	Register whose value is updated whenever a timing violation occurs. You can use notifiers to define responses to timing violations. (See “ <a href="#">Using Notifiers</a> ” on page 723 for details.)

If you specify more than one edge control specifier, the edges must be either all positive (01, 0X, X1) or all negative (10, 1X, X0).

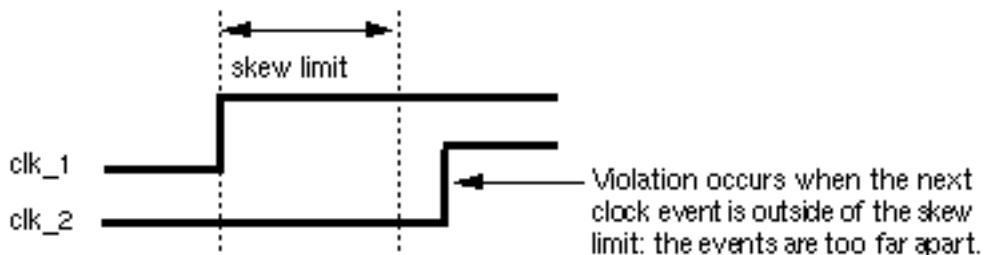
Example:

In the following example, the \$period system task reports a violation if the time between the reference event (negedge clk) and the implicit data event (the next negedge clk) is less than period\_param (13).

```
specify
  specparam period_param=13;
  $period( negedge clk, period_param ) ;
endspecify
```

## \$skew

The `$skew` system task specifies the maximum delay allowable between two signals. A violation occurs when signals are too far apart. The following figure shows how a violation occurs with the `$skew` system task.



The `$skew` system task has the following format:

```
$skew(reference_event, data_event, limit [,notifier]);
```

The `$skew` system task arguments are as follows:

---

\$skew Arguments	Description
<code>reference_event</code>	Module input or inout transition at a control signal that establishes the reference time for tracking violations on the data event.
<code>data_event</code>	Module input or inout transition at a signal that initiates the timing check against the value in <code>limit</code> .
<code>limit</code>	Positive constant expression or specparam that specifies the delay allowed between the transitions of the two signals.
<code>notifier</code> (optional)	Register whose value is updated whenever a timing violation occurs. You can use notifiers to define responses to timing violations. (See “ <a href="#">Using Notifiers</a> ” on page 723 for details.)

---

The `$skew` system task records the new time of the reference event before it performs the timing check. If the reference event and the data event occur at the same time, `$skew` does not report a timing violation.

Example:

In the following example, the `$skew` system task reports a violation if the interval from the reference event (`posedge clk1`) to the data event (`negedge clk2`) exceeds `skew_param` (14).

```
specify
  specparam skew_param=14;
  $skew(posedge clk1, negedge clk2, skew_param);
endspecify
```

## \$recovery

The `$recovery` system task specifies a time constraint between an asynchronous control signal and a clock signal (for example, between clearbar and the clock for a flip-flop). A violation occurs when a change in either signal occurs within the specified time constraint.

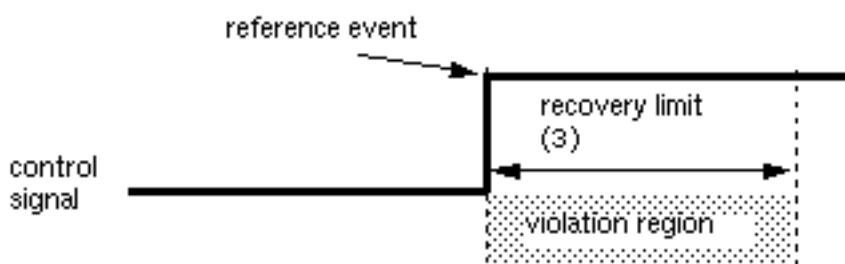
The `$recovery` system task has the following format:

```
$recovery(reference_event, data_event, recovery_limit [,notifier]);
```

The following `$recovery` system task reports a timing violation if the data event (`posedge clk`) occurs within the interval specified by `recovery_param` (3).

```
specify
  specparam recovery_param=3;
  $recovery( posedge set, posedge clk, recovery_param );
endspecify
```

The following figure shows the violation region:



The \$recovery system task arguments are as follows:

<b>\$recovery Arguments</b>	<b>Description</b>
<i>reference_event</i>	Asynchronous control signal, which normally has an edge identifier associated with it to indicate which transition corresponds to the release from the active state.
<i>data_event</i>	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.
<i>recovery_limit</i>	A positive minimum interval between the release of the asynchronous control signal and the next active edge of the clock.
<i>notifier</i> (optional)	Register whose value is updated whenever a timing violation occurs. You can use notifiers to define responses to timing violations. (See “ <a href="#">Using Notifiers</a> ” on page 723 for details.)

\$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.

## **\$removal**

The \$removal system task specifies a time constraint between an asynchronous control signal and a clock signal (for example, between clearbar and the clock for a flip-flop). A violation occurs when a change to either signal occurs within the specified time constraint.

The \$removal system task has the following format:

```
$removal(reference_event, data_event, removal_limit [,notifier]);
```

The following \$removal system task reports a timing violation if the data event (posedge clk) occurs within the interval specified by removal\_param (3).

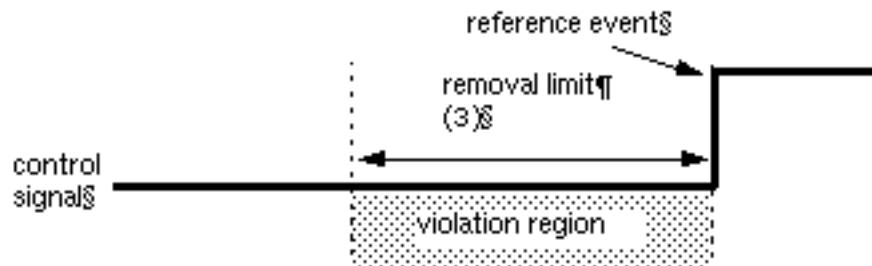
```
specify
  specparam removal_param=3;
  $removal( posedge set, posedge clk, removal_param );
endspecify
```

## NC-Verilog Simulator Help

### Timing Checks

---

The following figure shows the violation region:



The \$removal system task arguments are as follows:

---

\$removal Arguments	Description
<i>reference_event</i>	Asynchronous control signal, which normally has an edge identifier associated with it to indicate which transition corresponds to the release from the active state.
<i>data_event</i>	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.
<i>removal_limit</i>	A positive minimum interval between the release of the asynchronous control signal and the next active edge of the clock.
<i>notifier</i> (optional)	Register whose value is updated whenever a timing violation occurs. You can use notifiers to define responses to timing violations. (See “ <a href="#">Using Notifiers</a> ” on page 723 for details.)

---

If the *data\_event* and the *reference\_event* occur simultaneously, \$removal performs the timing check before it records the new *reference\_event* time. Therefore, no violation is reported.

## \$recrem

The \$recrem system task combines the functionality of \$recovery and \$removal 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 a change in one of the signals causes a violation of the specified constraint.

Two limits, corresponding to the removal and the recovery time constraints, must be specified. One of these limits can be negative, but the sum of the two limits must be 0 or greater.

The \$recrem system task has the following format:

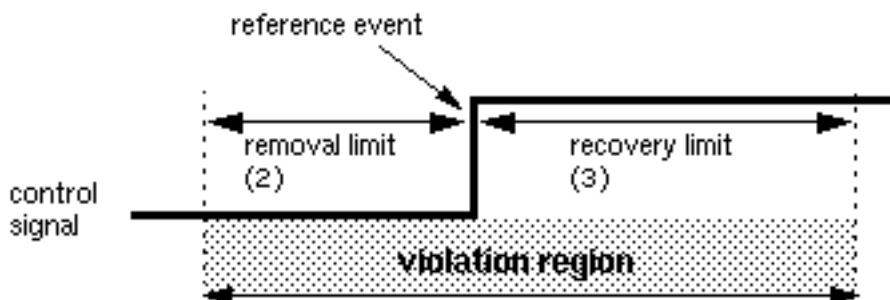
```
$recrem(control_event, data_event, recovery_limit, removal_limit,  
        [notifier], [tstamp_cond], [tcheck_cond],  
        [delayed_clk], [delayed_data]);
```

**Note:** Absent optional parameters must be indicated as null parameters by using commas. Do not add one or more commas after the last argument because the syntax can be truncated after any argument.

The following \$recrem system task contains two positive time limits. In this example, \$recrem reports a violation if the interval between posedge ctrl and clk is less than the value of tREC (which is 3), enacting its \$recovery component. It also reports a violation if the interval between posedge ctrl and clk is less than the value of tREM (which is 2), enacting its \$removal component.

```
specify  
  specparam tREC=3, tREM=2;  
  $recrem( posedge ctrl, clk, tREC, tREM );  
endspecify
```

The following figure shows the violation region:



## NC-Verilog Simulator Help

### Timing Checks

---

The `$recrem` system task arguments are as follows:

<b>\$recrem Arguments</b>	<b>Description</b>
<code>reference_event</code>	Asynchronous control signal, which normally has an edge identifier to indicate which transition corresponds to the release from the active state.
<code>data_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>recovery_limit</code>	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.
<code>removal_limit</code>	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.
<code>notifier</code> (optional)	Register whose value is updated whenever a timing violation occurs. You can use notifiers to define responses to timing violations. (See “ <a href="#">Using Notifiers</a> ” on page 723 for details.)
<code>tstamp_cond</code> (optional)	Places a condition on the <code>stamp_event</code> . For the removal check of <code>\$recrem</code> , this argument places a condition on the transition of the data signal. For the recovery check of <code>\$recrem</code> , this argument places a condition on the transition of the reference signal.
<code>tcheck_cond</code> (optional)	Places a condition on the <code>check_event</code> . For the removal check of <code>\$recrem</code> , this argument places a condition on the transition of the reference signal. For the recovery check of <code>\$recrem</code> , this argument places a condition on the transition of the data signal.
<code>delayed_clk</code> (optional)	Delayed signal value for <code>reference_event</code> when one of the limits is negative. See “ <a href="#">Negative Timing Check Limits in \$setuphold and \$recrem</a> ” on page 728 for more information.
<code>delayed_data</code> (optional)	Delayed signal value for <code>data_event</code> when one of the limits is negative. See “ <a href="#">Negative Timing Check Limits in \$setuphold and \$recrem</a> ” on page 728 for more information.

---

**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]

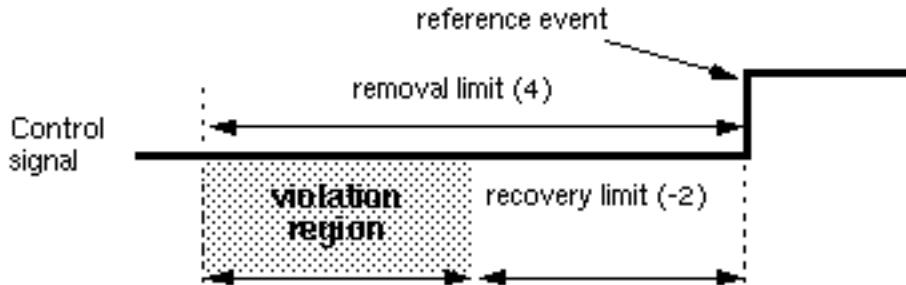
```
"/net/machine/home/willy/1.8/code/cond8.18.525", 31: $recrem(ckin && cond2in,  
datin, su, hl, flag, , condlin);
```

You can specify negative times for either the *recovery\_limit* or the *removal\_limit* argument. The sum of the two arguments must be 0 or greater. The following examples show the effects of negative values.

### Negative *recovery\_limit*

A negative *recovery\_limit* value specifies a time period preceding a change in the control signal. The following example shows a \$recrem system task with a recovery limit of -2 and a removal limit of 4.

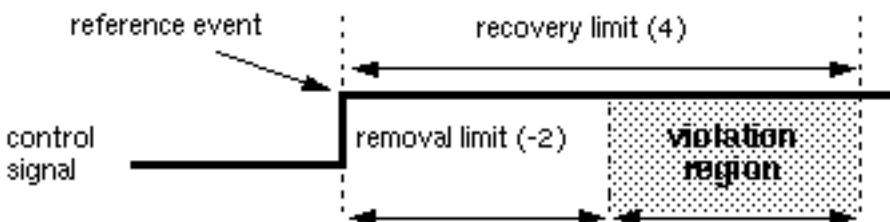
```
$recrem( posedge ctrl, clk,-2, 4, notifier );
```



### Negative *removal\_limit*

A negative *removal\_limit* value specifies a time period following a change in the control event signal. The following example shows the \$recrem system task with a recovery limit of 4 and a removal limit of -2.

```
$recrem( posedge ctrl, clk, 4, -2, notifier );
```



**Note:** Beginning with the LDV 3.3 release, the use of negative time specifications is enabled by default. With LDV 3.2 or a previous release, you must use the `-neg_tchk` option on the command line when you invoke the elaborator if you use negative time specifications. If you do not specify this 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.

## **\$nochange**

The `$nochange` system task is supported by Veritime. The NC-Verilog simulator compiles source descriptions containing calls inside specify blocks to `$nochange`, but ignores these calls during elaboration and simulation.

## **Using Edge-Control Specifiers**

You can control timing check events using specific edge transitions between 0, 1, and x. Edge-control specifiers begin with the keyword `edge` followed by a list of from one to six pairs of the following edge transitions:

01	From 0 to 1
0x	From 0 to x
10	From 1 to 0
1x	From 1 to x
x0	From x to 0
x1	From x to 1

The edge transitions are separated by commas, and the list is enclosed in square brackets. For example,

```
edge[01, 0x]
```

Edge transitions involving z are treated the same way as edge transitions involving x.

You can also use 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]`.

Example:

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 clk transitions 0->1 or x->1. Timing checks for \$width occur when clk transitions 1->0, x->0, or 1->x.

```
module DFF2(clk, d, q, qb);
input clk, d;
output q,b;
...
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);
    $hold(edge[01, x1]clk,d, tHold);
    $width(negedge clk, tWneg);
endspecify
endmodule
```

## Using Notifiers

Timing check notifiers let you detect timing check violations behaviorally, and take an action 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.

A notifier is a register that you pass as the last argument to a system timing check. The register must be declared in the module where the timing check tasks are invoked. The notifier is an optional argument that can be omitted from the system task call without affecting its operation.

If a notifier is included as the last argument, a timing violation will toggle the notifier's value as shown in the following table:

Before timing violation	After timing violation
x	1
0	1
1	0

## NC-Verilog Simulator Help

### Timing Checks

---

Before timing violation	After timing violation
z	z

**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 that the UDP received its inputs at time 0.

To specify that you want to ignore notifiers when performing timing checks, use the `-nonotifier` option when you elaborate the design.

```
% ncelab -nonotifier top_mod
```

The following examples show timing checks with notifier arguments.

**Example 1:**

```
$setup( data, posedge clk, 10, notify_reg );
$width( posedge clk, 16, flag );
```

**Example 2:**

The following is an example of how to use notifiers in a behavioral model. In this example, 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.

## NC-Verilog Simulator Help

### Timing Checks

---

```
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 ;
        ?    ?     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.

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]
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 can be deterministic, as in `==`, `!=`, `~`, or no operation; or non-deterministic as in `=`, or `!=`.

## NC-Verilog Simulator Help

### Timing Checks

---

When comparisons are deterministic, an `x` value on the conditioning signal will not enable the timing check. For non-deterministic comparisons, an `x` on the conditioning signal will enable the timing check.

A `scalar_expression` evaluating to `1'bz` is always true.

A `scalar_constant` evaluating to `1'bz` is interpreted as `1'b1`.

For compatibility with Verilog-XL, the `scalar_expression` should be a scalar net. The NC-Verilog simulator, however, accepts a general expression that can be a single-bit or multi-bit quantity. When the `scalar_expression` evaluates to a multi-bit quantity, the least significant bit is used.

A `scalar_constant` should be a single-bit constant value or expression, but if a multi-bit constant expression is used, only the least significant bit is considered. For example, the following expression is legal in NC-Verilog:

```
wire[3:0] A;  
&&&(A == 4'b1010)
```

In this case, the least significant bit of `A` and `4'b1010` are considered with the non-deterministic `==` operator.

**Example 1:**

The following example `$setup` timing check is unconditioned. The timing check will occur whenever there is a positive edge on `clk`.

```
$setup( data, posedge clk, 10 );
```

To trigger this timing check on the positive edge of `clk`, but only when `clr` is high, rewrite the command as follows:

```
$setup( data, posedge clk &&& clr, 10 );
```

To trigger this timing check on the positive edge of `clk`, but only when `clr` is low, use one of the following commands:

```
$setup( data, posedge clk &&& (~clr), 10 );  
$setup( data, posedge clk &&& (clr==0), 10 );
```

**Example 2:**

If you want to condition a timing check using multiple conditioning signals, you can add a statement outside the specify block to create a gate whose output is then used as 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. Add the following declaration outside the specify block:

```
and( clr_and_set, clr, set );
```

2. Add the condition to the timing check, using the signal `clr_and_set` as follows:

```
$setup( data, posedge clk && clr_and_set, 10 );
```

## Negative Timing Check Limits in `$setuphold` and `$recrem`

The use of negative time specifications in `$setuphold` or `$recrem` timing checks is enabled by default. Use the `-noneg_tchk` option when you invoke the elaborator to disallow the use of negative values. If you use this option, negative limits are set to 0 in the description or annotation, and a warning is issued.

**Note:** With LDV 3.2 or a previous release, you must use the `-neg_tchk` option on the command line when you invoke the elaborator in order for the simulator to use the negative limit values. If you do not specify this option, negative limits are set to 0 in the description or annotation, and a warning is issued..

Using negative limits in these timing checks can affect the evaluation of timing checks. For each timing check with a negative limit, the reference and/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 729 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 731 for details on how delay values are calculated and on how limits are adjusted.

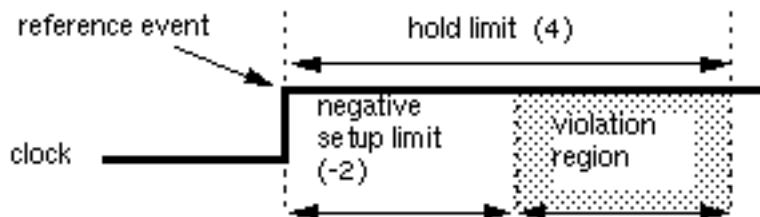
In some cases, you may want to drive your functional model using the delayed version of signals. To do this, you can explicitly define the delayed versions of signals in the `$setuphold` and `$recrem` timing checks using the `delayed_reference` and `delayed_data` arguments. See “[Explicitly Defining Delayed Signals](#)” on page 739 for details.

## Effects of Delayed Signals on Timing Checks

When a negative limit value is specified in a `$setuphold` or `$recrm` 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.

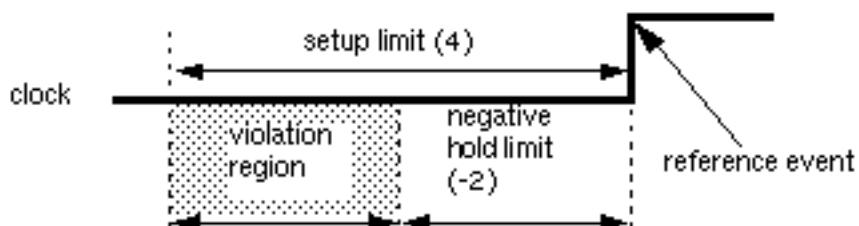
Example 1:

```
$setuphold(posedge clk, d -2, 4);
```



Example 2:

```
$setuphold(posedge clk, d, 4, -2);
```



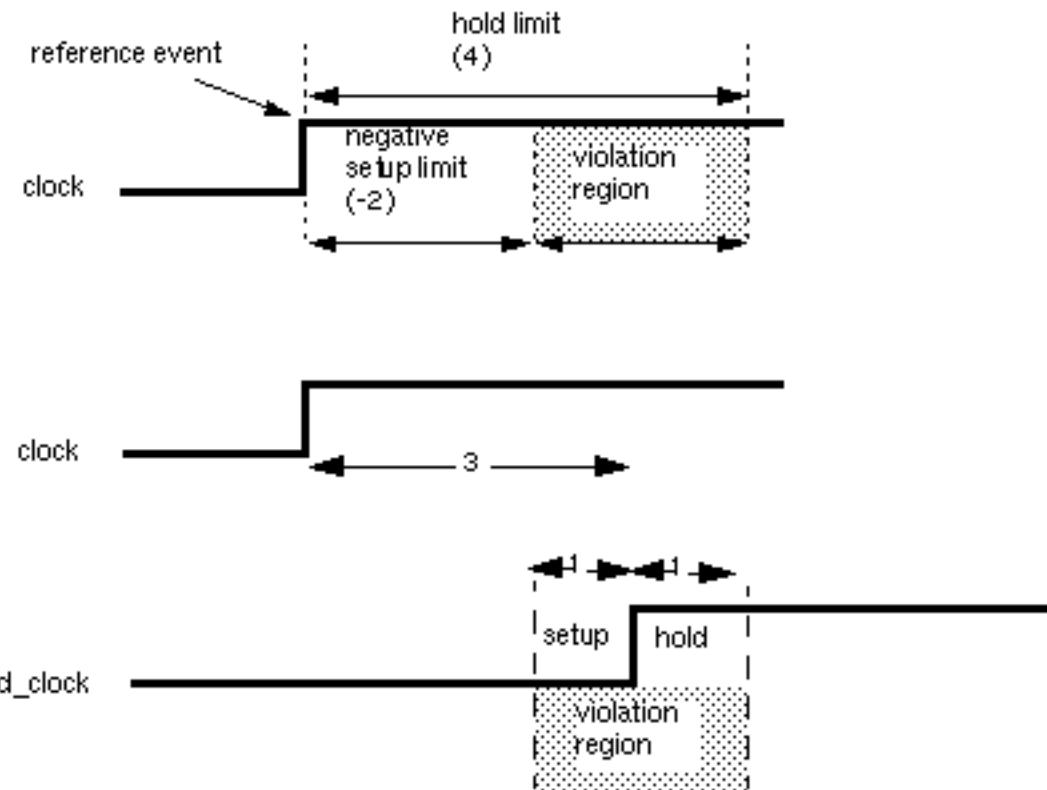
Because the violation region no longer extends from the reference signal, the constraints cannot be verified when the check events occur in the same way that they can be without negative limits. In the case of `$hold`, only the stamp events that occur after the value of the time offset (2 in the case of our example) should be considered. In the case of `$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.

## NC-Verilog Simulator Help

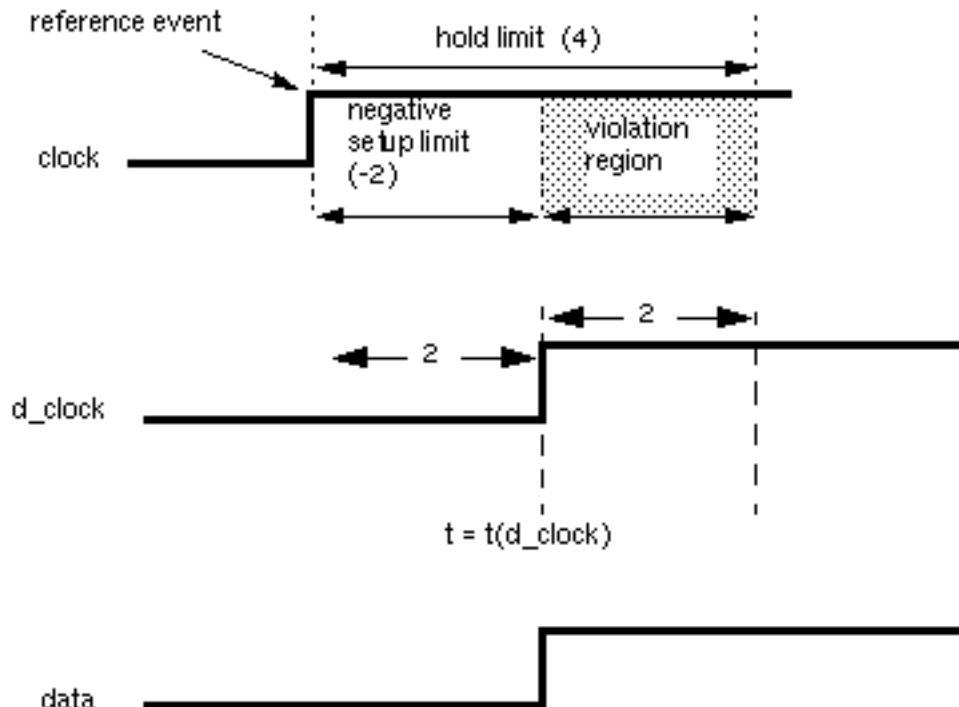
### Timing Checks

For example, the following figure shows the violation region for the `$setuphold` timing check shown in Example 1 above. To verify the negative setup constraint 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.

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. The NC-Verilog simulator uses the smallest simulation precision to determine this additional increment.

See “[Calculation of Delayed Signals and Limit Modification](#)” on page 731 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 check does not have negative limits. For example, if multiple timing checks are driven with the

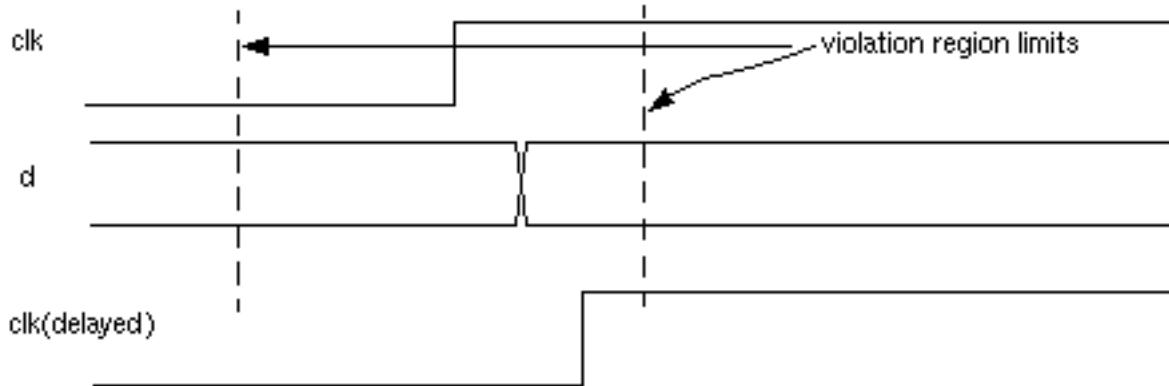
signal CLK, then one delay is calculated for CLK, and each timing check is evaluated using this single delayed version of CLK.

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, the NC-Verilog simulator 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.

## NC-Verilog Simulator Help

### 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;
    foreach setup/hold/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.

## Non-Convergence in Timing Checks

In some cases, negative values specified in \$setuphold or \$recrem timing checks, or in SDF SETUPHOLD or RECREM constructs, do not get annotated correctly. The SDF log file indicates that correct values are annotated, but negative values have been set to 0 and unexpected timing violations are reported. This problem is caused when the negative timing check algorithm cannot converge because all setup/hold times must be adjusted to positive and non-zero values.

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))
```

But because the values cannot be negative or zero, the algorithm does not converge and, if you use the ncelab -ntc\_warn command-line option (ncverilog +ntc\_warn), non-convergence warnings such as the following are issued:

```
ncelab: *W,NTCNN: Non-convergence of negative timing check values in instance  
test.shift_0.out_reg.
```

The algorithm then tries to force 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, the

## NC-Verilog Simulator Help

### Timing Checks

---

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.

For example, consider the timing checks in the following example:

```
'timescale 1ns/1ps
module test;
    reg data, clock;
    wire q;

initial
begin
    $monitor( $time, " clock = %b data = %b q = %b", clock, data, q);
    fork
        #10 data = 0;
        #10 clock = 0;
        #13 data = 1'b1;
        #14 clock = 1'b1;
        #30 clock = 1'b0;
        #38.5 data = 1'b0;
        #40 clock = 1'b1;
    join
    #10 $finish;
end

dEdgeFF u1 ( q,clock, data);

endmodule

module dEdgeFF(q, clock, data);
    output q;
    reg q;
    input clock, data;

initial
#10 q = 0;

always
@(negedge clock)#10 q=data;
```

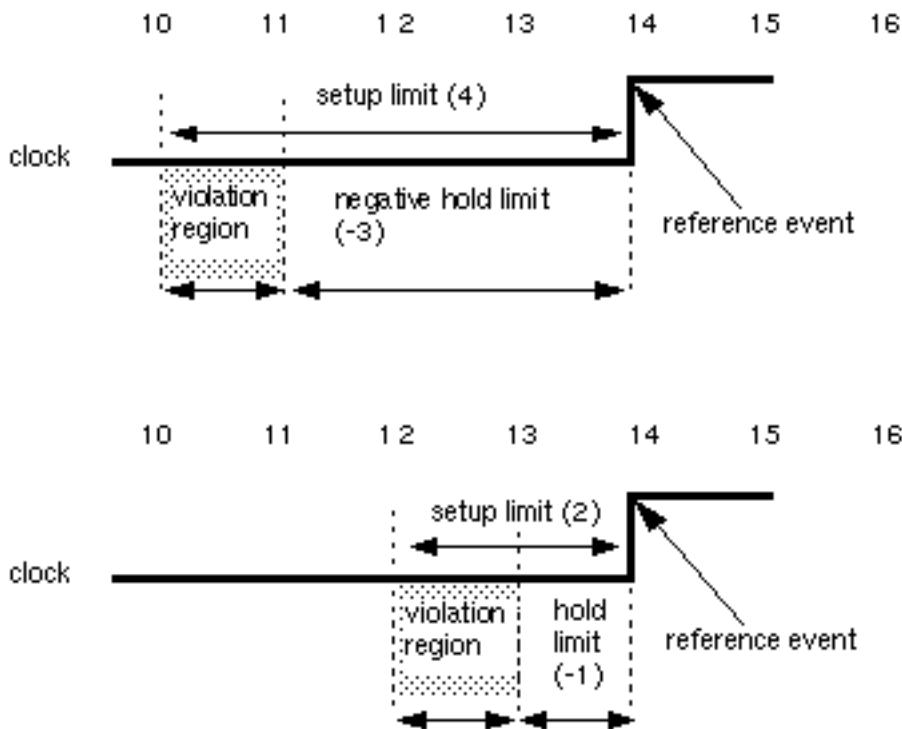
```

specify
    $setuphold(posedge clock, posedge data, 4, -3);
    $setuphold(posedge clock, negedge data, 2, -1);
endspecify

endmodule

```

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, causes the negative timing check algorithm not to converge. 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 so the tool can correctly place the delayed clock (that is, converge). Otherwise, it must set the negative value to 0.

When you elaborate this model with the `-ntc_warn` option, non-convergence warning messages (NTCNNC) are generated, and both negative hold values are set to zero. The two timing checks, in effect, are now as follows:

## NC-Verilog Simulator Help

### Timing Checks

---

```
$setuphold(posedge clock, posedge data, 4, 0);  
$setuphold(posedge clock, negedge data, 2, 0);
```

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:

```
Warning! Timing violation  
$setuphold<setup>( posedge clock:14 NS, posedge data:13 NS,  
        4.000 : 4 NS, 0.000 : 0 FS );  
File: ./ff.v, line = 45  
Scope: test.ul  
Time: 14 NS
```

```
Warning! Timing violation  
$setuphold<setup>( posedge clock:40 NS, negedge data:38500 PS,  
        2.000 : 2 NS, 0.000 : 0 FS );  
File: ./ff.v, line = 46  
Scope: test.ul  
Time: 40 NS
```

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 one base simulation time unit.

For example, you could edit the timing checks in the example to change the -3 hold time on the positive edge to -1.998. This makes the violation regions overlap by two 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 `ncelab -extend_tcheck_data_limit` or `-extend_tcheck_reference_limit` command-line options.

These options automatically extend the violation regions by a specified percentage so that they overlap.

Syntax:

```
-extend_tcheck_data_limit percent_relaxation  
-extend_tcheck_reference_limit percent_relaxation
```

If you are running *ncverilog*, the corresponding command-line options are `+ncextend_tcheck_data_limit/percent_relaxation` and `+ncextend_tcheck_reference_limit/percent_relaxation`.

The *percent\_relaxation* argument is the maximum percentage increase allowed in the timing violation window to achieve an overlap of two time precision units.

The `-extend_tcheck_data_limit` option changes the hold or recovery limit in the timing checks so that the violation regions overlap by at least two units of simulation precision. 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.

### **Example 1:**

In the previous example, the timing checks are:

```
$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:

```
% ncelab -extend_tcheck_data_limit 100 worklib.test:module
```

This command extends the violation region created by the first timing check by 1.002 to create some overlap.

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 ns with the following command:

```
% ncelab -extend_tcheck_reference_limit 100 worklib.test:module
```

### **Example 2:**

Suppose that you have the following pair of timing checks, where there are two non-overlapping violation windows, each with a width of 2 ns.

```
$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.

```
% ncelab -extend_tcheck_reference_limit 100 worklib.test:module
```

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:

```
% ncelab -extend_tcheck_data_limit 50 worklib.test:module
```

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.

You can specify only one of these options on the command line. Using these options automatically turns on the `-ntc_warn` option.

When you use either of these options, and if the specified relaxation percentage allows the timing checks to converge, the elaborator issues a warning message (NTCRLX) to let you know 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 NC-Verilog 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, 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/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.

## NC-Verilog Simulator Help

### Timing Checks

---

```
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
```

The NC-Verilog simulator 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
```

The NC-Verilog simulator 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

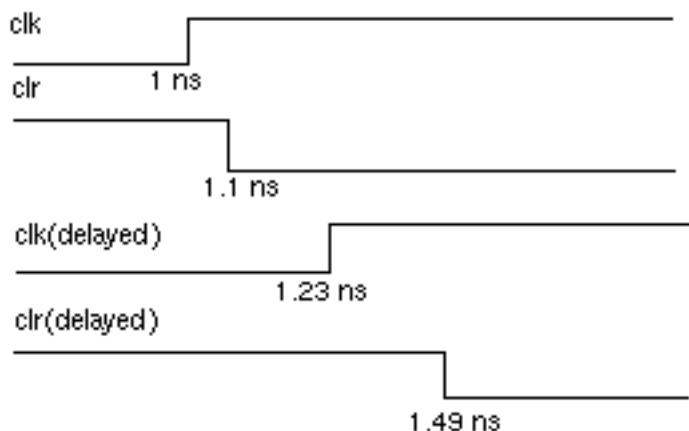
Delayed signals may affect path delays. When you specify a negative timing check, NC-Verilog chooses a path delay that may be different from the path delay that is chosen without negative timing checks.

To illustrate this, consider the following set of timing checks and path delays and the input waveforms (with delayed signals) in the following figure.

## NC-Verilog Simulator Help

### Timing Checks

```
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.

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 because the actual device probably follows the shorter `clr` delay.

The NC-Verilog simulator 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.

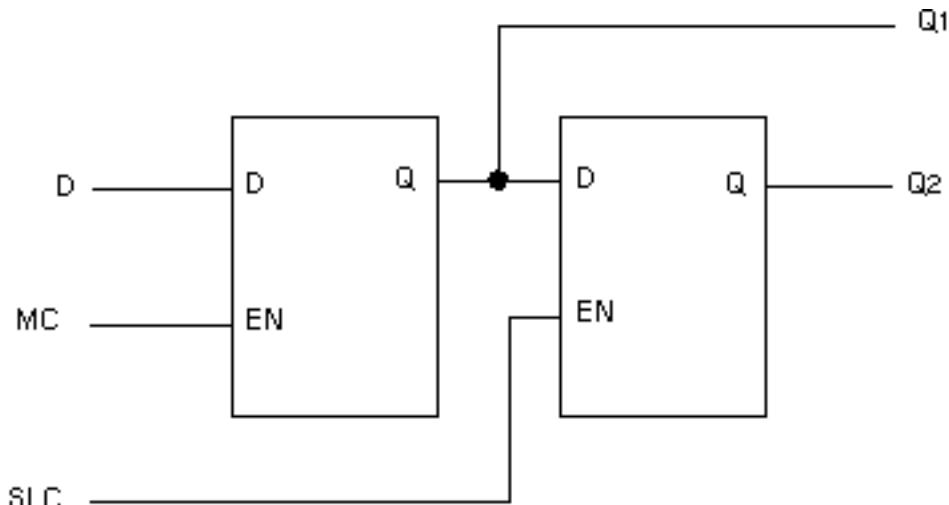
The delay calculated for a delayed signal should not be longer than a path delay with that signal as a source. After the delays for the delayed signals are calculated, all path delays in a module are scanned, and if any are longer than the delayed signal for their source, a warning is issued if the `-ntc_warn` 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 731.

## 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.
- 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, NC-Verilog 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 NC-Verilog uses this approximation algorithm by specifying the `-ntc_warn` option on the command line. By default, a warning is not printed.

## Timing Violation Messages

When a system timing check encounters a timing violation, NC-Verilog reports the following information:

- Time of the second event (the violation)
- Time of the first event
- Value of the timing check limit
- File and line number
- Instance name of the module in which the violation occurred
- Time of the violation

Timing check violation messages have one of two different formats depending on whether `'timescale` compiler directives control the modules containing them. In both of the

## NC-Verilog Simulator Help

### Timing Checks

---

examples shown below, a timing violation occurred on line 13 of the Verilog source description file ff.v in board.counter.a.

The following message shows that without the `timescale directive the \$setup system task reported the violation that occurred at time 60 NS with a clock time of 100 and a timing check limit of 50.

```
Warning! Timing violation
$setup( data: 60 NS, posedge clock: 100 NS, 50 : 50 NS);
File: ./ff.v, line = 13
Scope: board.counter.a
Time: 100 NS + 1
```

The following example shows that with the `timescale directive the \$setup system task reported the violation that occurred at time 60 US with a clock time of 100 US and a timing check limit of 50. The 50 in the violation message is the unscaled value that appears in the timing check code; the 50 US is the scaled value that the timing check tests.

```
Warning! Timing violation
$setup( data: 60 US, posedge clock: 100 US, 50 : 50 US);
File: ./ff.v, line = 13
Scope: board.counter.a
Time: 100 US + 1
```

The values of time limits in timing violation messages are always current, reflecting any changes made by PLI and SDF annotation.

When a timing violation occurs due to a vector, system timing checks report one violation for each bit that changed.

Violation messages for \$setuphold are similar to other violation messages, but they also include information on which component of the timing check was violated. The following example shows a typical \$setuphold violation message:

```
Warning! Timing violation
$setuphold <setup> (posedge clock: 100 NS, data: 60 NS, 50
50 NS, 50 : 50 NS);
File: ./ff.v, line = 13
Scope: board.counter.a
Time: 100 NS + 1
```

A violation of \$setuphold in signals passing from a vector port to a vector port generates an identical message for each bit that experiences a violation.

If you want the time values in the violation messages to be printed using the same formatting rules that Verilog-XL uses, use the -xlstyle units option when you invoke *ncsim* or

change the value of the `display_unit` predefined variable to `xlstyle` after you have invoked the simulator (See “[Setting Variables](#)” on page 481).

## SDF Annotation of Timing Checks

You can annotate timing check information using an SDF file. Use the SDF file `TIMINGCHECK` keyword and associated timing check constructs. (See “[TIMINGCHECK Keyword and Constructs](#)” on page 1025 for details.)

Timing checks in the SDF file are mapped to corresponding HDL constructs as follows:

- A timing check with no edges or no conditions in the SDF file, matches any timing check with the same arguments.
- A timing check with an edge in the SDF file must have the same edge in the HDL.
- A timing check with a condition in the SDF file must have the same condition in the HDL.

## Referencing Verilog HDL Source Constructs

The NC-Verilog simulator lets you annotate objects at the source level or select “sub-paths” in the SDF file. For example, if you have the following Verilog description:

```
wire [3:0] A, Y;  
specify  
    $setuphold( A, Y, 3, 2);  
endspecify
```

An SDF file can contain the following constructs:

```
(SETUPHOLD A Y (10) (11))  
(SETUPHOLD A[3] Y[3] (8) (9))
```

The first statement references the Verilog HDL source constructs at the source level, and the second statement references the constructs at the bit level. There is no restriction that all references must be made at the source or bit-level. In addition, source and bit-level references can be made to the same statement.

This is different from Verilog-XL, which requires that you specify whether timing checks in `specify` blocks are to be expanded or unexpanded using the `+expand_specify_vectors` command line option or the ``expand_specify_vectors` and ``noexpand_specify_vectors` compiler directives.

## **\$setuphold Timing Checks**

A **SETUP** in the SDF file modifies the setup limit in a matching **\$setuphold**, and a **HOLD** modifies the hold limit. A **SETUPHOLD** in the SDF file can be used to modify both limits in a **\$setuphold**, or it can be used to annotate separate **\$setup** and **\$hold** checks.

The following are all valid annotations in NC-Verilog:

### **Example1:**

Verilog:

```
$setup( D, posedge clk, 10 );
$hold( posedge clk, D, 10 );
```

SDF:

```
( SETUP D (posedge clk) (5) )
( HOLD D (posedge clk) (5) )
```

### **Example 2:**

Verilog:

```
$setup( D, posedge clk, 10 );
$hold( posedge clk, D, 10 );
```

SDF:

```
( SETUPHOLD D (posedge clk) (5) (6) )
```

### **Example 3:**

Verilog:

```
$setuphold( posedge clk, D, 10, 10);
```

SDF:

```
( SETUPHOLD D (posedge clk) (5) (6) )
```

### **Example 4:**

Verilog:

```
$setuphold( posedge clk, D, 10, 10);
```

## NC-Verilog Simulator Help

### Timing Checks

---

SDF:

```
( SETUP D (posedge clk) (5) )
( HOLD D (posedge clk) (6) )
```

With negative setup or hold limits, the same flexibility applies. However, after a cell has been annotated, all \$setuphold limits are checked to make sure that the sum of the two limits is 0 or greater. In Verilog-XL, the +splitsuh option is implied, and only SETUPHOLD is allowed in the SDF file and verified.

See [Chapter 17, “SDF Timing Annotation.”](#) for more information on SDF annotation.

## Interconnect and Module Path Delays

---

This chapter contains the following sections:

- [Interconnect Delays](#)
- [Module Path Delays](#)

### Interconnect Delays

The wire connecting source ports to load ports is responsible for interconnect delays. A full interconnect delay specification describes the delay from the output of one module to the input of another module, while a port interconnect delay specification describes the delay to that port from all source ports connected to it.

Annotation of interconnect delays in the NC-Verilog simulator is similar to annotation in Verilog-XL. You can annotate interconnect delays using SDF or PLI routines.

While NC-Verilog does not include the Verilog-XL concepts of Module Input Port Delays (MIPDs), Single-source Interconnect Transport Delays (SITDs), or Multi-source Interconnect Transport Delays (MITDs), the default type of interconnect delay behaves like a Module Input Port Delay (MIPD) in Verilog-XL. Delays are inertial.

In Verilog-XL, you use two command-line plus options together, `+multisource_int_delays` and `+transport_int_delays`, to enable transport delay behavior with pulse control and the ability to specify unique delays for each source-load path. In the NC-Verilog simulator, you use the `-intermod_path` command-line option when you invoke the elaborator (`ncelab`) to enable this functionality.

**Note:** `ncverilog` and `ncprep` automatically translate the Verilog-XL options to `-intermod_path`.

## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

---

The following table summarizes the default type of interconnect delay and interconnect delays with the `-intermod_path` option:

<b>Aspect</b>	<b>Default Interconnect</b>	<b>With <code>-intermod_path</code></b>
Ports subject to delay	One set of delays specified to each port. No unique source-load delays.	Unique source-load delays can be specified.
Transitions affected	Twelve transitions.  Verilog-XL allows three: to 1, to 0, to Z	Twelve transitions.  Verilog-XL allows six: 0->1, 1->0, 0->Z, Z->0, 1->Z, Z->1
min:typ:max triplets	Available for each of the twelve delays.	Available for each of the twelve delays.
Delay handling	Inertial delay.	Transport delay.
Pulse control	None.	Global and per-annotation pulse control available.
Observability of signal values by system tasks	Post-interconnect delay monitoring available. Delayed signal value is visible at annotated load port.  In Verilog-XL, the delayed signal value is visible only after it passes through a primitive.	Post-interconnect delay monitoring available. Delayed signal value is visible at annotated load port.

## Default Interconnect Delays

This section describes the default type of interconnect delay in more detail.

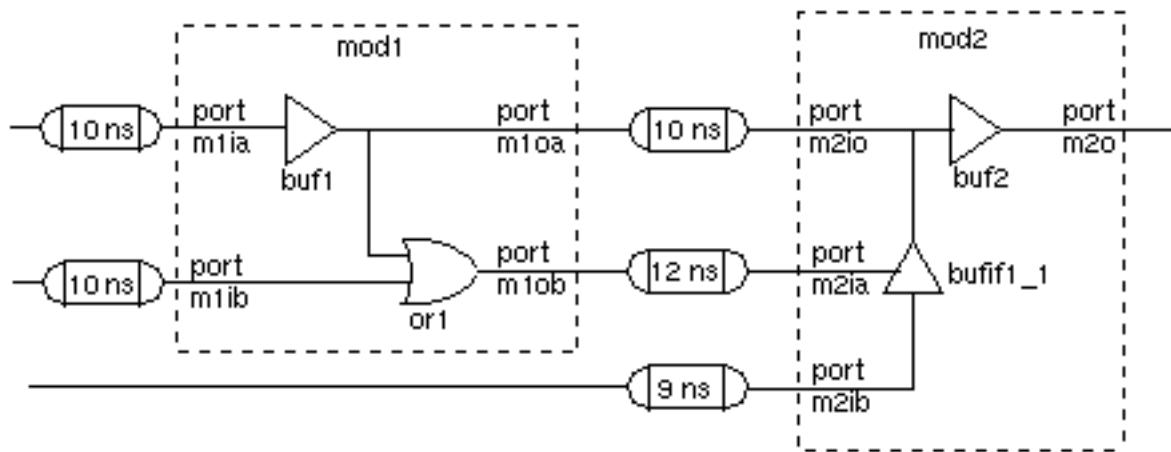
- Interconnect delays are specified only on input and inout ports. In the following figure, the valid ports for interconnect delays are `m1ia`, `m1ib`, `m2io`, `m2ia`, and `m2ib`. The

## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

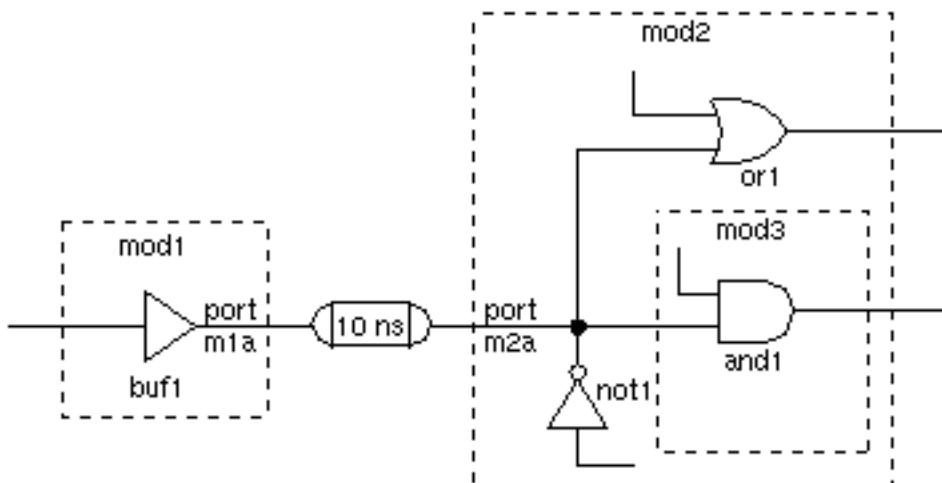
---

interconnect delays have been explicitly represented as delay lines for purposes of illustration.



- Delays affect all levels of the hierarchy. Delays are distributed to each load in a port's fanout. They require no specific hierarchical relationship between drivers and loads or between ports and loads. Delays are distributed to loads within the port's module and down to lower hierarchical levels. The propagation of transitions is delayed 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 an interconnect delay:



In this figure, an interconnect delay placed on input port `m2a` of module `mod2` provides a delay between port `m1a` of module `mod1` and port `m2a` of module `mod2`. All loads of port `m2a` inside module `mod2` also see this delay.

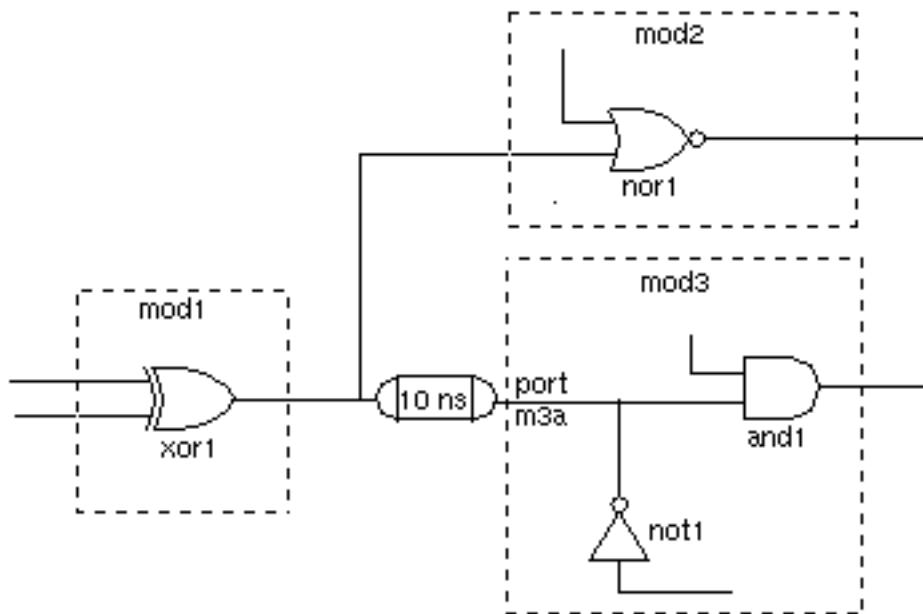
## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

Notice that the net has two sources: the `buf1` in `mod1` and the `not1` in `mod2`. You can also specify an interconnect delay from `not1` to `or1` and `and1`, assuming that these are modules that have named ports.

- Delays are directional. You can specify a full source-load delay in either direction through an inout port. A port delay is always interpreted as a delay into a port from sources outside the module. Port delays cannot be annotated to outputs.

In the following figure, the delay to port `m3a` of `mod3` specifies a delay from driver `xor1` to the load `and1`, whether this port is declared as an input or as an inout. It does *not* specify a delay to `nor1` in module `mod2`.

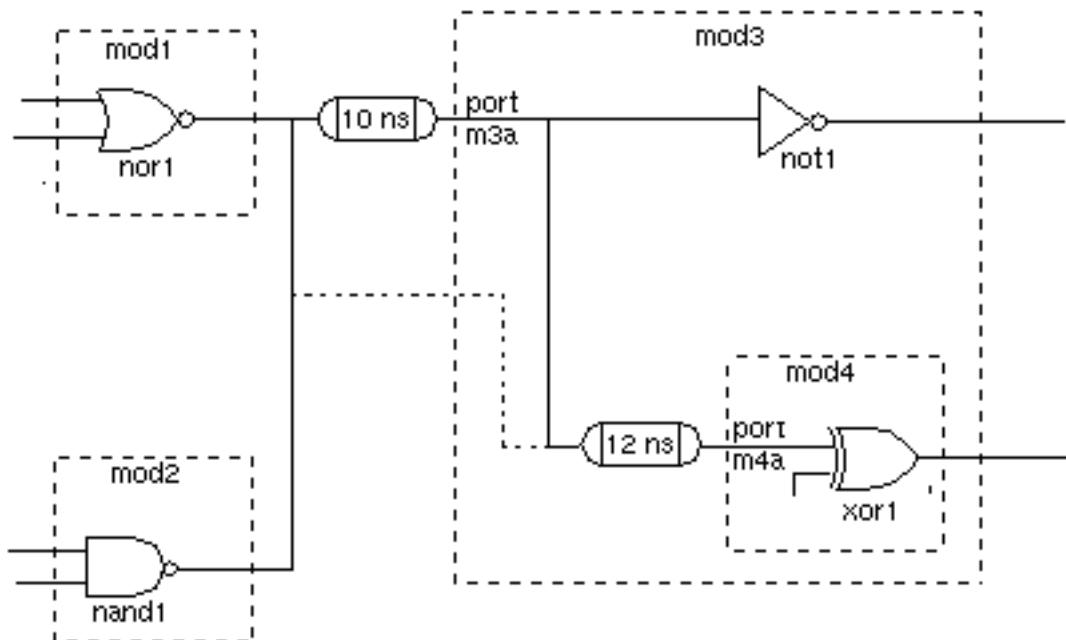


- For paths with multiple sources, you can fully specify source-load interconnect delays by using the `-intermod_path` option when you invoke the elaborator.

## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

The following figure shows modules with single-bit ports and their valid interconnect delays.



In this figure, port `m3a` of module `mod3` and port `m4a` of module `mod4` can each have their own unique delay from each source. In the figure, the dotted line to the interconnect delay on port `m4a` indicates that the sources for it are actually `nor1` and `nand1`, not port `m3a`. If you specify a delay only to port `m3a`, the delay to port `m4a` will be the same as the delay to `m3a`.

- Each bit in a port can have only one delay. If a port comprises more than one bit, you can assign a different delay for each bit.

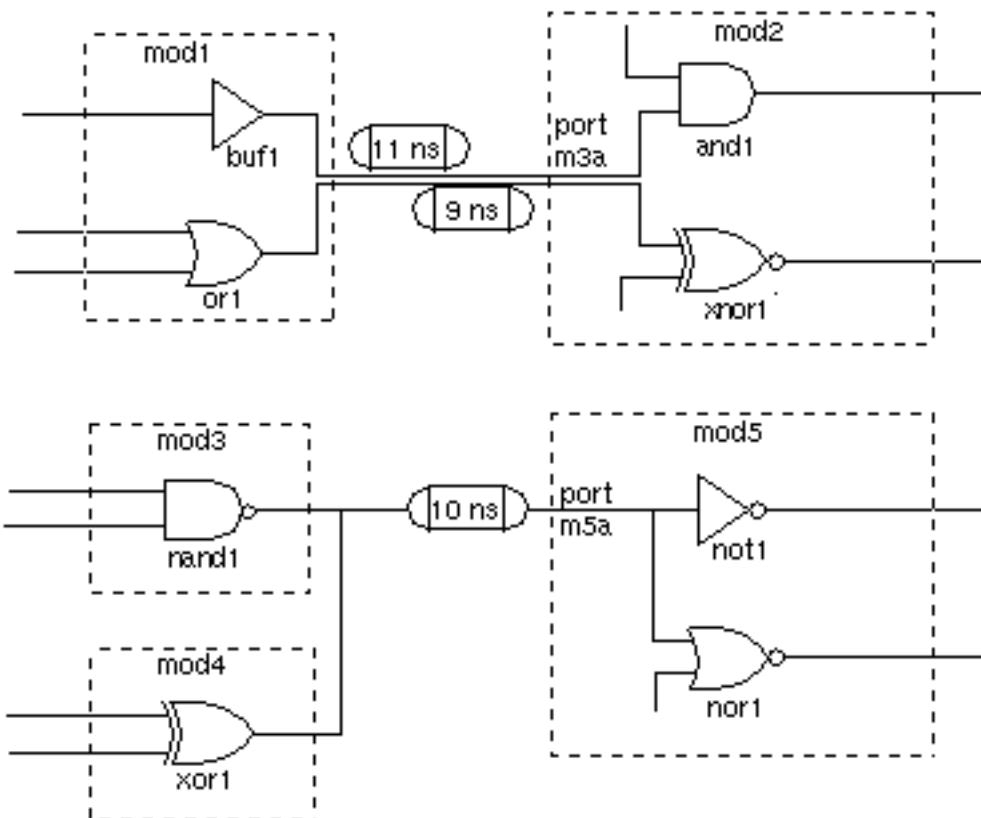
The following figure shows the valid delays for multiple-bit and single-bit ports. In this figure, port `m3a` was declared to be 2 bits wide. One bit connects `buf1` to `and1`, and the other bit connects `or1` to `xnor1`. You must specify a delay for each bit of the port

## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

---

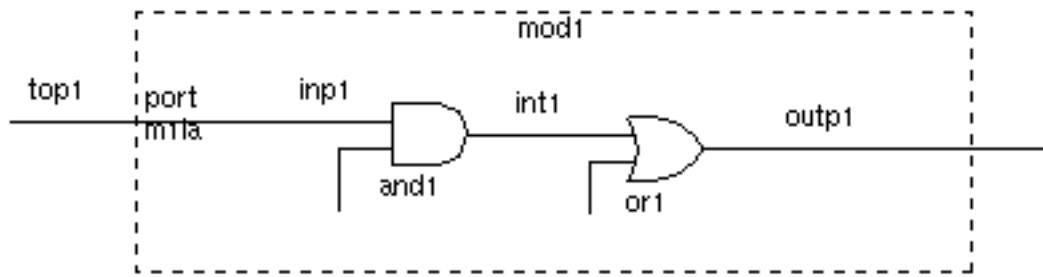
individually. The NC-Verilog simulator does not permit the annotation of more than one bit at a time.



- The default interconnect delays are inertial delays just like the delays on primitives. An inertial delay filters out pulse widths shorter than the specified delay. If a pulse width is exactly as long as a delay, you cannot predict if NC-Verilog will schedule or filter out the pulse width.
- Interconnect delays affect timing checks. If a timing check's data or reference event propagates through a port with a delay, the event is delayed by the amount of the delay. If this delay means the event is no longer within the time limit of the timing check, the timing check does not report a timing violation.
- Strength changes are propagated through interconnect delays, but are not affected by them.
- In the NC-Verilog simulator, if you annotate an interconnect from a source to a load, monitoring the source or any net "before" the load yields undelayed signal values. Monitoring the net at or "after" the load yields the delayed signal value. For example, in the following figure, nets top1 and inp1 are connected by port m1ia, and an

interconnect delay has been annotated to port `m1ia`. Monitoring the port or `inp1` displays delayed signal values.

In Verilog-XL, the delay element is associated with the primitive it drives, and any monitoring of the net yields undelayed signal values. For this example, Verilog-XL displays the same values and transition times for `top1` and `inp1`.



The same distinction applies to behavioral statements. If `inp1` is used in a behavioral statement in NC-Verilog, the delayed value is used. In Verilog-XL, the undelayed value is used.

## Interconnect Delays and -intermod\_path

Use the `-intermod_path` option when you invoke the elaborator (`ncelab`) to enable transport delay behavior, pulse control, and unique delays for source-load paths. By using this option, you can impose delays from the same source to different loads, and you can specify different delays and pulse limits on paths from different sources to the same load.

## Pulse Handling

By default, pulse control limits are set to the delay to yield inertial delay behavior. To see transport delay behavior, you must set pulse control limits:

- Use the `-pulse_r` and the `-pulse_e` options when you invoke the elaborator. These options set global pulse limits for both module path delays and interconnect delays.
- Use the `-pulse_int_r` and `-pulse_int_e` options to set limits for interconnect delays only. For interconnect delays, these options take precedence over settings for `-pulse_r` and `-pulse_e`.
- Specify reject and error limits in the SDF file constructs.

## SDF Annotation of Interconnect Delays

Use the SDF PORT, INTERCONNECT, or NETDELAY constructs to annotate interconnect delays. See “[PORT Keyword](#)” on page 1014, “[INTERCONNECT Keyword](#)” on page 1016, and “[NETDELAY Keyword](#)” on page 1018 for details on these SDF file constructs.

See [Chapter 17, “SDF Timing Annotation,”](#) for details on SDF annotation.

## PLI Annotation of Interconnect Delays

Use the PLI/VPI `acc_replace_delays`, `acc_append_delays`, and `vpi_put_delays` routines to modify delays at simulation time.

You must use the `-anno_simtime` option when you elaborate the design to enable the use of these routines. If you do not specify this option at elaboration time, and a PLI/VPI routine that modifies delays is executed at simulation time, the simulator issues a message and the delay modification does not take place.

## Module Path Delays

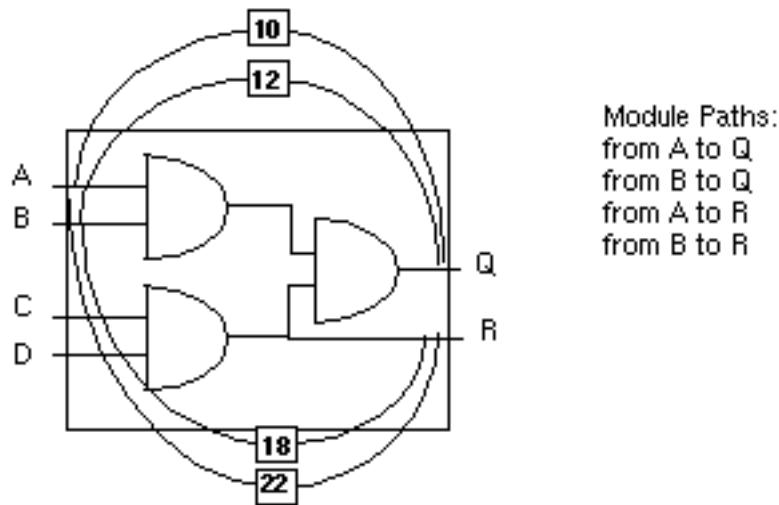
The Verilog HDL can describe two types of delays:

- *distributed delays*, which specify the time it takes events to propagate through gates and nets inside the module.
- *module path delays*, which specify the time it takes an event at a source (input port or inout port) to propagate to a destination (output port or inout port).

This section tells you how to describe module paths and how to assign delays to those paths.

Module paths pair a signal source with a signal destination. The module source signal can be unidirectional (an input port) or bidirectional (an inout port). The module destination signal

can be unidirectional (an output port) or bidirectional (an inout port). The following figure illustrates a circuit with module path delays:



Module path delays are described inside specify blocks (see “[Specify Blocks](#)” on page 758 for details).

There are three aspects to defining module path delays in a specify block:

1. Describe the module paths (see “[Describing Module Paths](#)” on page 759 for details).

Module paths can be described as:

- Simple paths
- Edge-sensitive paths
- State-dependent paths

2. Decide if the connection between the source and destination is to be a full or a parallel connection path (see “[Establishing Full or Parallel Connection Paths](#)” on page 768 for details).

3. Assign delays to the paths (see “[Assigning Delays to Module Paths](#)” on page 771 for details).

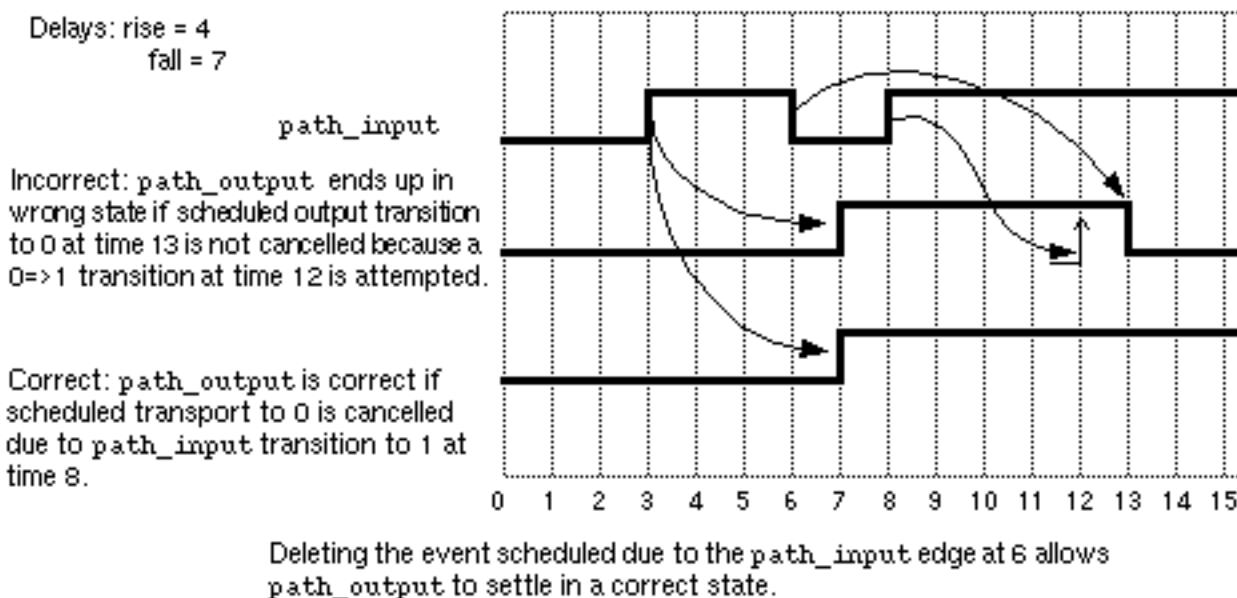
In the NC-Verilog simulator, module path delays (and interconnect delays) are simulated as transport delays by default. No command-line option is necessary to enable the transport delay algorithm. You must, however, set pulse control limits in order for pulses that have a shorter duration than the delay to pass through to the output. If you do not set pulse control limits, the limits are set equal to the delay by default. This means that, even though module path delays and interconnect delays are simulated as transport delays, with no pulse control

## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

the results look as if they are being simulated as inertial delays. (See “[Setting Pulse Controls](#)” on page 280 for details.)

In addition to pulses being lost or filtered due to pulse control, pulses can also be filtered due to event cancellation. 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.



As this figure shows, passing all transitions in transport delay would make the output transmit an incorrect signal. NC-Verilog, like Verilog-XL, deletes scheduled events that would lead to such a result.

## Specify Blocks

A specify block is a block statement in which you describe paths across a module and assign delays to those paths. In addition to describing module path delays, the specify block can also be used for:

- Performing timing checks to ensure that events occurring at the module inputs satisfy the timing constraints of the device described by the module. (See [Chapter 15, “Timing Checks,”](#) for details.)
- Defining pulse filtering limits for a specific module or for particular paths within a module. (See [“Setting Pulse Controls”](#) on page 280 for details.)

The syntax for a specify block is as follows:

```
specify_block ::= specify [specify_item] endspecify
specify_item ::=  
    specparam_declaration  
    |= path_declaration  
    |= system_timing_check
```

Example:

```
specify  
    // Two specparam declarations  
    specparam tRise_clk_q = 150, tFall_clk_q = 200;  
    specparam tSetup = 70;  
    // Module path delay  
    (clk => q) = (tRise_clk_q, tFall_clk_q);  
    // System timing check  
    $setup(d, posedge clk, tSetup);  
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 syntax for declaring specify parameters is as follows.

```
specparam_declaration ::= specparam list_of_specparam_assignments;  
list_of_specparam_assignments ::= specparam_assignment { , specparam_assignment }  
specparam_assignment ::=
```

```
identifier = constant_expression  
| pulse_control_specparam
```

The value assigned to a specify parameter can be any constant expression, and a specify parameter declared in the specify block can be used to construct a constant expression for a subsequent specify parameter declaration.

Example:

```
specify  
    specparam tRise_clk_q=150, tFall_clk_q=200;  
    specparam tRise_control=40, tFall_control=50;  
endspecify
```

Specify parameters and module parameters are not interchangeable. The following table summarizes the differences between the two types of parameter declarations.

Specparams	Parameters
Use keyword <code>specparam</code>	Use keyword <code>parameter</code>
Declared inside specify blocks	Declared outside specify blocks
Used only inside specify blocks	Not used in specify blocks
Cannot use <code>defparam</code> to override values	Use <code>defparam</code> to override values

Specify blocks cannot appear in macro modules.

## Describing Module Paths

A module path is described inside a specify block as a connection between a source signal and a destination signal. The following restrictions apply to the source and the destination:

- The module path source must be a net that is connected to a module input or inout port.
- The module path destination must be a net or register that is connected to a module output or inout port.

Module paths can connect any combination of vectors and scalars.

More than one source can have a module path to the same output, and different delays can be specified for each input to output path.

The IEEE 1364 standard (*IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*) states that a module path can be described as:

- A simple path
- An edge-sensitive path
- A state-dependent path

The syntax of the module path declaration is described in the IEEE standard as follows.

```
path_declaration ::=  
    simple_path_declaration;  
  | edge_sensitive_path_declaration;  
  | state_dependent_path_declaration;
```

Cadence has extended this syntax to include four specify properties that enhance the path delay selection algorithm. These properties give you more control over the selection of a delay when there are multiple inputs that occur either simultaneously or while a path delay output is already scheduled. These extensions are discussed in [“Specify Properties for Module Path Delays”](#) on page 776.

## Simple Module Paths

A simple module path is a path with no edge-sensitive or state-dependent conditions. The syntax of a simple module path declaration is as follows:

```
simple_path_declaration ::=  
    parallel_path_description = path_delay_value;  
  | full_path_description = path_delay_value;  
  
parallel_path_description ::=  
    (specify_input_terminal_descriptor [polarity_operator] =>  
     specify_output_terminal_descriptor);  
  
full_path_description ::=  
    (list_of_path_inputs [polarity_operator] *-> 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 [msb_constant_expression:lsb_constant_expression]

specify_output_terminal_descriptor ::=

    output_identifier
    | output_identifier [constant_expression]
    | output_identifier [msb_constant_expression:lsb_constant_expression]

input_identifier ::= input_port_identifier | inout_port_identifier

output_identifier ::= output_port_identifier | inout_port_identifier
```

The syntax shows the two operators you can use in a path declaration:

- \* $>$  establishes a full connection between the source and the destination (see “[Full Connections](#)” on page 768 for details).
- = $>$  establishes a parallel connection between the source and the destination (see “[Parallel Connections](#)” on page 769 for details).

Examples:

```
(A *> Q) = 10;
(B => Q) = (12);
(C, D *> Q) = 18;
```

## Edge-Sensitive Module Paths

The IEEE 1364 standard describes an edge-sensitive module path delay as a module path that is described using an edge transition at the source. The syntax of an edge-sensitive module path declaration is shown in the IEEE specification as follows:

```
edge_sensitive_path_declaration ::=

    parallel_edge_sensitive_path_description = path_delay_value;
    | full_edge_sensitive_path_description = path_delay_value;

parallel_edge_sensitive_path_description ::=

    ([edge_identifier] specify_input_terminal_descriptor =>
        (specify_output_terminal_descriptor
            [polarity_operator] : data_source_expression))

full_edge_sensitive_path_description ::=

    ([edge_identifier] list_of_path_inputs *->
        (list_of_path_outputs
            [polarity_operator]: data_source_expression))

data_source_expression ::= expression
```

## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

---

The edge identifier can be the keyword `posedge` or `negedge`. The optional polarity operator describes whether the data path is inverting or noninverting. The data source expression describes the flow of data to the path destination.

These constructs are used by timing analyzers. The NC-Verilog simulator, like Verilog-XL, compiles source code containing this syntax, but ignores the edge identifier, polarity operators, and the data source expression.

Example:

The following example shows how NC-Verilog interprets module path descriptions that contain edge keywords. The upper specify block contains valid syntax, but NC-Verilog interprets these module paths as if they were the module path descriptions in the lower specify block.

```
specify
    specparam trise=2, tfall=3;
    (posedge clock => (out1 -: in3)) = (trise, tfall);
    (negedge clock => (out2 +: in3)) = (trise, tfall);
endspecify
```

```
specify
    specparam trise=2, tfall=3;
    (clock => out1) = (trise, tfall);
    (clock => out2) = (trise, tfall);
endspecify
```

You can implement the functionality of edge keywords using state-dependent path delays. (See “[State-Dependent Module Paths](#)” on page 763 for details.) For example, the following lines attempt to impose different edge-conditioned delays on a path:

```
(posedge clk => (q_out +: d_in)) = 15;
(negedge clk => (q_out +: d_in)) = 9;
```

However, the NC-Verilog simulator does not use the edge information. It interprets the preceding lines as follows and chooses 9, the smaller of the delays.

```
(clk => q_out) = 15;
(clk => q_out) = 9;
```

The following lines implement the intended edge conditions.

```
if (clk == 1) (clk => q_out) = 15;
if (clk == 0) (clk => q_out) = 9;
```

## State-Dependent Module Paths

A state-dependent module path delay (SDPD) is a conditional module path delay. It assigns a delay to a module path when specific conditions are true. The syntax of an SDPD is as follows:

```
<state_dependent_path_declaration> ::=  
    if (conditional_expression) simple_path_declaration;  
    | if (conditional_expression) edge_sensitive_path_declaration;  
    | ifnone simple_path_declaration;
```

The conditional expression must evaluate to true for the path to be assigned a delay value. Expressions that evaluate to 1, X, or Z are treated as true. If the result is multi-bit, the lsb represents the result.

**Note:** The evaluation of SDPD conditional expressions is unlike the evaluation of other Verilog HDL constructs. For example, in the behavioral language, if statements that evaluate to X or Z are treated as false.

If multiple SDPDs are specified for a path, NC-Verilog looks at the delays for all statements whose condition is true and whose source had the most recent transition, and then selects the smallest delay. This behavior is the same as Verilog-XL.

The operands in an SDPD conditional expression must be one of the following:

- Scalar or vector module input or inout port in its entirety or in bit-select or part-select form.
- Compile time constant (constant numbers or specify parameters).
- Parameter (an expression that can be changed after compile time). The updated value is *not* used. The parameter value used is the compile time value.
- Net or register declared within the module containing the SDPD description.

The SDPD conditional expression can have any number of operators, and all operators are valid.

You can use edge keywords (posedge and negedge) in state-dependent module path descriptions, but NC-Verilog ignores their meaning. Polarity operators and data source expressions are also ignored. For example, the following path description:

```
if (!reset && !clear) (posedge clock => (out +: in)) = (10, 8);
```

is interpreted as follows:

```
if (!reset && !clear) (clock => out) = (10, 8);
```

See “[Edge-Sensitive Module Paths](#)” on page 761 for more information.

## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

---

Use the `ifnone` keyword to specify a default path delay for cases where all of the SDPD conditional expressions are false. You cannot specify an `ifnone` condition for a path and an unconditional simple path for the same module path.

#### Example 1:

In the following example, the first two SDPDs describe a pair of output rise and fall delay times when the XOR gate (`x1`) 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 x1 (out,a,b);
specify
    specparam noninvrise = 1, noninvfall = 2
    specparam invertrise = 3, invertfall = 4;
    if(a) (b=>out)=(invertrise,invertfall);
    if(b) (a=>out)=(invertrise,invertfall);
    if(~a)(b=>out)=(noninvrise,noninvfall);
    if(~b)(a=>out)=(noninvrise,noninvfall);
endspecify
endmodule
```

#### Example 2:

In the following example, SDPDs specify different sets of path delays for different ALU operations. The first three path declarations declare paths extending from operand inputs `i1` and `i2` to the `o1` output. The delays on these paths are assigned to operations on the basis of the operation specified by the inputs on `opcode`. The last path declaration declares a path from the `opcode` input to the `o1` output.

```
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)
```

## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

---

```
(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
```

#### Example 3:

The following example includes an `ifnone` condition, which specifies a default path delay for cases where all of the SDPD conditional expressions are false:

```
if (c1) (in => out) = 10;
if (c2) (in => out) = 9;
ifnone (in => out) = 8;
```

#### Example 4:

The following is another example of using the `ifnone` keyword to specify a default path delay for cases where all of the SDPD conditional expressions are false:

```
// 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);

// all other operations
ifnone (i2 => o1) = (15.0, 15.0);
```

#### Example 5:

The following example is illegal because it combines an SDPD using an `ifnone` condition and an unconditional path for the same module path:

```
if (a) (b => out) = (2, 2);
if (b) (a => out) = (2, 2);
```

## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

---

```
ifnone (a => out) = (1, 1);
(a => out) = (1, 1);
```

#### Unknowns on Level-Sensitive Delays

With the typical implementation of level-sensitive qualifiers, NC-Verilog 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, both conditional expressions evaluate as true, and 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, NC-Verilog treats the condition as 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
if (flag == 1)(in => out) = a;	Yes	No	Yes
if (flag == 0)(in => out) = b;	No	Yes	Yes
if (flag == X)(in => out) = c;	Yes	Yes	Yes
(in => out) = d;	Yes	Yes	Yes
Delay selected for path from in to out	min (a,c,d)	min (b,c,d)	min (a,b,c,d)

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. If you do care about the value of `flag`, specify a complete set of conditional path delays (a and b).

The minimum delays are used because the only time when multiple paths should be selected is when unknowns are introduced into the conditional expressions. When unknowns are involved as part of the conditional expression, then it is likely that the output value will be corrupted by the unknown signal. This will result in the output signal going to an unknown value after the minimal delay.

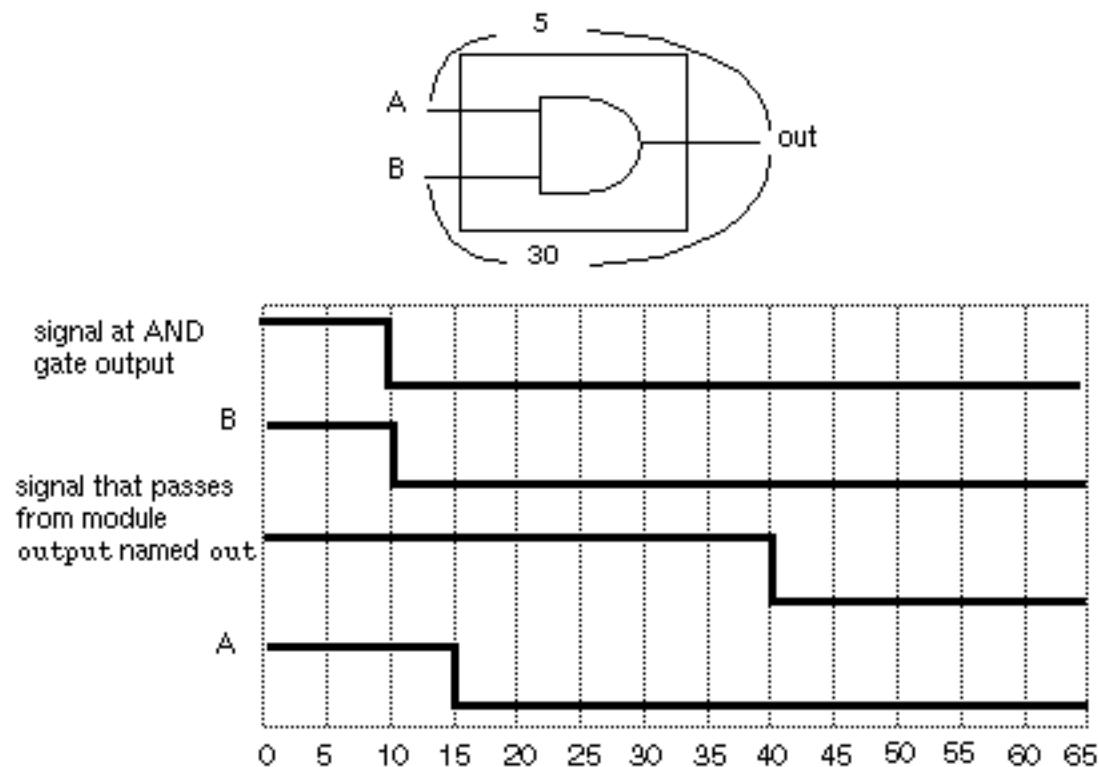
## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

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.

#### 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, NC-Verilog evaluates the gate and determines 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, NC-Verilog does not reschedule `out`'s change to time 20, because the simulator schedules output changes when edges transmit to module outputs. `A`'s change 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 `B`'s change 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.

## Establishing Full or Parallel Connection Paths

Two types of connections can be established between the source and the destination when describing a module path: full or parallel.

### Full Connections

The `*>` operator establishes a *full* connection between source and destination. A full connection establishes a connection between every bit in the source and 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 establish a full connection 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.

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. For example, the following statement:

```
(a, b, c *> q1, q2) = 10;
```

Is equivalent to the following six individual module path assignments:

```
(a *> q1) = 10;  
(a *> q2) = 10;  
(b *> q1) = 10;  
(b *> q2) = 10;  
(c *> q1) = 10;  
(c *> q2) = 10;
```

## Parallel Connections

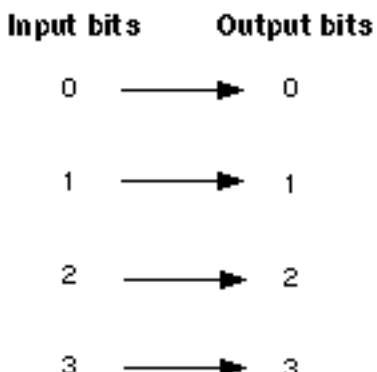
The `=>` operator establishes a *parallel* connection between source and destination. A parallel connection establishes a connection between each bit in the source to each corresponding bit in the destination.

Parallel module paths can be created only between one source and one destination where each signal contains the same number of bits. That is, a parallel connection is used to describe a path between two vectors of the same size. Since scalars are one bit wide, either `*>` or `=>` can be used to set up bit-to-bit connections between two scalars.

### Example 1:

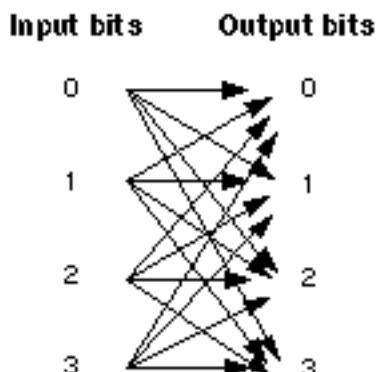
The following figure illustrates how a parallel connection differs from a full connection between two 4-bit vectors.

**Parallel module path**



4 paths  
Bit-to-bit connections  
Use `=>` to define path

**Full module path**



16 paths  
Bit-to-vector connections  
Use `*>` to define path

### Example 2:

In the following example, the module path from `s` to `q` uses a full connection (`*>`) 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 a parallel connection (`=>`) because they set up parallel connections between two 8-bit busses.

```
module MUX8 (in1, in2, s, q);
input [7:0] in1, in2;
input s;
output [7:0] q;

...
...

specify
    (in1 => q) = (3, 4);      // parallel connection
    (in2 => q) = (2, 3);      // parallel connection
    (s *> q) = 1;            // full connection
endspecify
endmodule
```

### Module Path Polarity

In the Verilog HDL, you can specify the polarity of a module path. The polarity indicates whether or not the direction of a signal transition is inverted as it propagates from the input to the output.

A module path can have:

- 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.

## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

---

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
```

Polarity has no effect on the scheduling of simulation events, but is used by some timing analyzers to calculate module path delays. The NC-Verilog simulator, like Verilog-XL, ignores all polarity operators.

## Assigning Delays to 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.

You specify delays as a list of one, two, three, six, or twelve path delay expressions separated by commas. You can specify a single delay value representing the typical delay, or a colon-separated list of three delay values for minimum, typical, and maximum delay. See [“Calculating Delay Values for X Transitions”](#) on page 774 for information on how delay values for x transitions are calculated when they are not explicitly specified.

---

Number of path delay expressions	Description
1	Specifies one delay for the following six transitions: 0->1 1->0 0->Z Z->1 1->Z Z->0

## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

---

<b>Number of path delay expressions</b>	<b>Description</b>
2	Specifies a rise delay for: 0->1, 0->Z, Z->1  Specifies a fall delay for: 1->0, 1->Z, Z->0
3	Specifies delays for: rising (0->1, Z->1) falling (1->0, Z->0) Z transitions (0->Z, 1->Z)
6	Specifies six different transition delays for the following six transitions: 0->1 1->0 0->Z Z->1 1->Z Z->0
12	Specifies delays for the six transitions shown above, plus delays for transitions to and from X. 0->X X->1 1->X X->0 X->Z Z->X

---

Examples:

The following examples show you how to specify delays using one delay value and three delay values in min:typ:max form:

```
// One delay expression
// Assign a delay of 20 for all transitions from C to Q
(C => Q) = 20;
```

## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

---

```
// Assign min:typ:max delays to all transitions from C to Q
(C => Q) = 10:14:20;

// Two delay expressions
// Assign rise and fall delays
specparam tPLH = 12, tPHL = 25;
(C => Q) = (tPLH, tPHL);
specparam tPLH = 12:16:22, tPHL = 16:22:25;
(C => Q) = (tPLH, tPHL);

// Three delay expressions
// Assign delays for rise, fall, and z transitions
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);

// Six delay expressions
// Assign delays for transitions to and from 0, 1, and z
specparam t01 = 12, t10 = 16, t0z = 13,
          tz1 = 10, t1z = 14, tz0 = 34;
(C => Q) = (t01, t10, t0z, tz1, t1z, tz0);
specparam t01=12:14:24, t10=16:18:20, t0z=13:16:30;
specparam tz1=10:12:16, t1z=14:23:36, tz0=15:19:34;
(C => Q) = (t01, t10, t0z, tz1, t1z, tz0);

// Twelve delay expressions
// Specify all transition delays explicitly
specparam t01 = 12, t10 = 16, t0z = 13,
          tz1 = 10, t1z = 14, tz0 = 34,
          t0x = 14, tx1 = 15, t1x = 15,
          tx0 = 14, txz = 20, tzx = 30;
(C => Q) = (t01, t10, t0z, tz1, t1z, tz0, t0x, tx1, t1x, tx0, txz, tzx);
```

In the following example, 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.

## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

---

```
specify
    specparam tRise_clk_q=45:150:270, tFall_clk_q=60:200:350;
    specparam tRise_control=35:40:45, tFall_control=40:50:65;
    (clk=>q)=(tRise_clk_q,tFall_clk_q);
    (clr,pre*>q)=(tRise_control,tFall_control);
endspecify
```

To specify that you want to use the minimum delays, use the `-mindelays` option on the command line when you elaborate the design.

```
% ncelab -mindelays top_module
```

Use the `-maxdelays` option for maximum delays and `-typical` (the default) for typical delays.

If you are running in single-step invocation mode using *ncverilog*, use `+maxdelays`, `+mindelays`, or `+typdelays`.

### Calculating Delay Values for X Transitions

If you do not explicitly specify X transition delays, the calculation of delay values for X transitions is based on the following two pessimistic rules:

- Transitions from a known state (s) to X ( $s \rightarrow X$ ) should occur as quickly as possible—that is, they receive the shortest possible delay.
- Transitions from X to a known state (s) ( $X \rightarrow s$ ) should 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.

---

<b>X Transition</b>	<b>Delay Value</b>
General Algorithm	
$s \rightarrow X$	Minimum of ( $s \rightarrow s$ )
$X \rightarrow s$	Maximum of ( $s \rightarrow s$ )
Specific Transitions	
$0 \rightarrow X$	Minimum of (0 $\rightarrow z$ delay, 0 $\rightarrow 1$ delay)
$1 \rightarrow X$	Minimum of (1 $\rightarrow z$ delay, 1 $\rightarrow 0$ delay)
$Z \rightarrow X$	Minimum of (z $\rightarrow 1$ delay, z $\rightarrow 0$ delay)

## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

---

X Transition	Delay Value
X -> 0	Maximum of (Z -> 0 delay, 1 -> 0 delay)
X -> 1	Maximum of (Z -> 1 delay, 0 -> 1 delay)
X -> Z	Maximum of (1 -> Z delay, 0 -> Z delay)
Usage: (C=>Q) = (5, 12, 17, 10, 6, 22)	
0 -> X	Minimum of (17, 5) = 5
1 -> X	Minimum of (6, 12) = 6
Z -> X	Minimum of (10, 22) = 10
X -> 0	Maximum of (22, 12) = 22
X -> 1	Maximum of (10, 5) = 10
X -> Z	Maximum of (6, 17) = 17

### Selecting a Delay When Multiple Delays Are Specified for a Path

The following table summarizes how the NC-Verilog simulator selects a delay from multiple path delay specifications. Verilog-XL selects delays in the same way as NC-Verilog.

Delay Types and Examples	NC-Verilog Delay Choice
Multiple unconditional path delays  (in => out) = 10; (in => out) = 9;	Error
Multiple unconditional path delays, one of which redefines an input or output as a bit- or part-select  (in => out) = 10; (in[1] => out[1]) = 9;	Error
Two edge-sensitive delays with different edges  (posedge in => (out:d1)) = 10; (negedge in => (out:d2)) = 9;	Ignore edge keywords and select min delay. delay = min(9, 10)

## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

---

Delay Types and Examples	NC-Verilog Delay Choice
Unconditional edge-sensitive delay and unconditional level-sensitive delay  (posedge in => (out:d)) = 10; (in => out) = 9;	Error
Multiple SDPDs  if (c1) (in => out) = 10; if (c2) (in => out) = 9;  Multiple SDPDs and an unconditional path delay  if (c1) (in => out) = 10; if (c2) (in => out) = 9; (in => out) = 8;	Select min of all true conditions.  if (c1 && c2) delay = min(10, 9) else if (c1) delay=10 else if (c2) delay=9 else delay=0;  Unconditional path delay is always true.  Select min of all true conditions.  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;
Multiple SDPDs and ifnone delay  if (c1) (in => out) = 10; if (c2) (in => out) = 9; ifnone (in => out) = 8;	Select min of all true conditions.  Select ifnone delay if no true conditions.  if (c1 && c2) delay=min(10,9) else if (c1 && !c2) delay=10 else if (!c1 && C2) delay=9 else delay = 8;

---

## Specify Properties for Module Path Delays

The default delay selection algorithm for module path delays can, under certain glitch conditions during simulation, result in the wrong delay or an undesired delay being selected. These problems tend to arise when modeling basic primitives such as NANDs and NORs. Cadence has extended the syntax of the specify block with four specify properties that provide more flexibility so that you can better control which delays are selected. These properties let you choose how a set of path delays operate when there are multiple inputs transitioning simultaneously or when a path delay output has already been scheduled.

The four specify properties are:

- `pathdelay_sense`
- `pathdelay_max0`

- `pathdelay_max1`
- `pathdelay_contROLSignal`

You can disable the enhanced timing features provided by these `specify` properties by using the `ncelab -disable_enht` command-line option.

### **pathdelay\_sense**

The `pathdelay_sense` `specify` property forces the transition time of a given path delay to be recalculated when one of its sensitive inputs changes value and an output is already scheduled.

Without `pathdelay_sense`, the transition time is recalculated only if a new output value is to be scheduled on the timing output. With `pathdelay_sense`, the output transition time is recalculated even if the same output value is to be used. This may allow a previously scheduled output to be rescheduled to a shorter transition time.

The syntax is as follows:

```
pathdelay_sense output [, input ...]
```

Specifying one or more inputs is optional. If you specify inputs, only those inputs are considered in the calculation. If you do not specify any inputs, the default is to use all of the inputs included in the module path delay specifications in the `specify` block to the specified output.

Example:

Suppose that you model the rise and fall delays of a NAND gate as follows:

```
nand (out, in1, in2);  
specify  
  (in1 *> out) = (30, 29);  
  (in2 *> out) = (10, 9);  
endspecify
```

Given the following input stimulus, the results are as follows:

Stimulus:

```
#0  in1 = 1; in2 = 1;  
#100 in1 = 0;  
#1  in2 = 0;
```

Results:

```
At time 0:  out = X  
At time 9:  out -> 0  
At time 130: out -> 1
```

- At time 9, `out` transitions to 0. This delay is selected because both `in1` and `in2` transitioned to 0 at time 0, and the algorithm selects the smallest fall delay (9).

## NC-Verilog Simulator Help

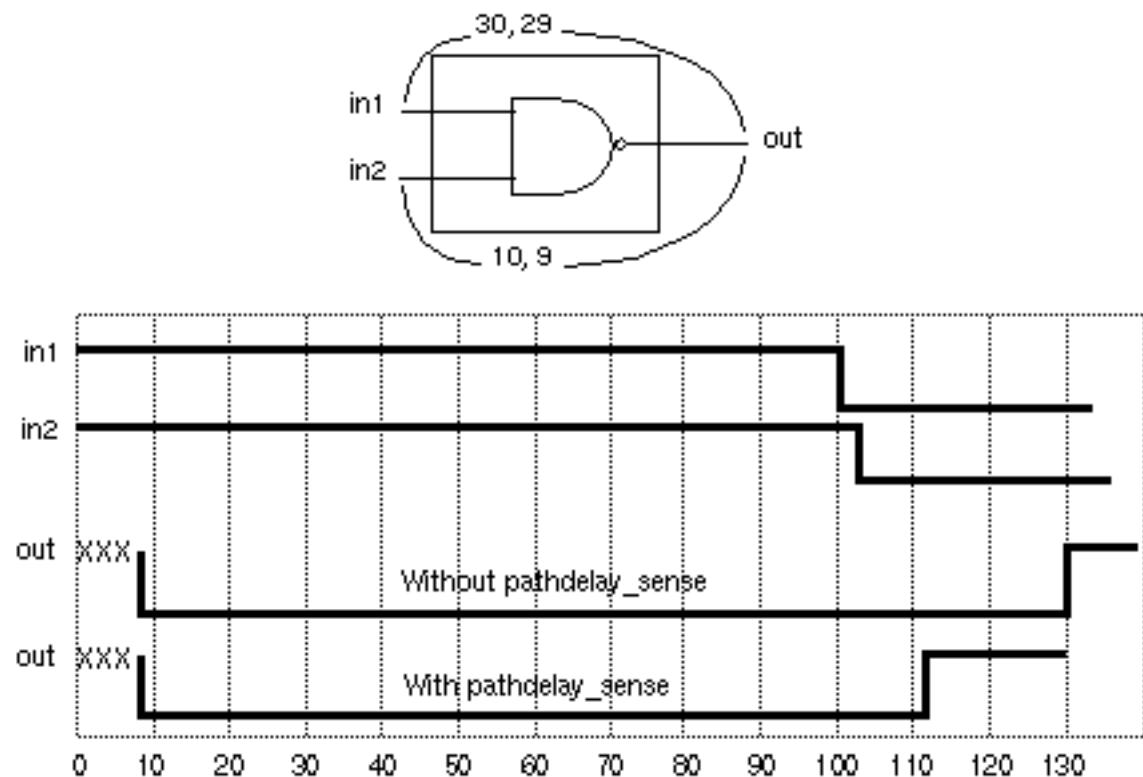
### Interconnect and Module Path Delays

- At time 100, `in1` transitions to 0. This schedules a transition on `out` to 1 at time 130.
- At time 101, `in2` transitions to 0. However, the transition time to 1 at 130 is not recalculated because, by default, the transition time is calculated only if a new output value is to be scheduled on the output.

Using the `pathdelay_sense` specify property, the output transition time is recalculated even if the same output value is to be used.

```
specify
  pathdelay_sense out, in1, in2;
  (in1 *> out) = (30, 29);
  (in2 *> out) = (10, 9);
endspecify
```

In this example, the `pathdelay_sense` property forces the recalculation of the output transition time when `in2` transitions to 0 at time 101. The transition time is rescheduled after a delay of 10 (that is, at time 111). The following figure illustrates the effect of using the `pathdelay_sense` property.



### **pathdelay\_max0 and pathdelay\_max1**

By default, the delay selection algorithm selects a delay by looking at the most recent transition(s), and if more than one input has occurred, selecting the smallest delay.

The `pathdelay_max0` property specifies that the longest transition time is to be selected when the output transitions to 0. The `pathdelay_max1` property specifies that the longest transition time is to be selected when the output transitions to 1. You can use both properties on the same output.

The syntax is as follows:

```
pathdelay_max0 output [, output ...]  
pathdelay_max1 output [, output ...]
```

#### **Example 1:**

Suppose that you model the rise and fall delays of a NAND gate as follows:

```
nand (out, in1, in2);  
specify  
    (in1 *> out) = (30, 29);  
    (in2 *> out) = (10, 9);  
endspecify
```

Given the following input stimulus, the results are as follows:

**Stimulus:**

```
#0    in1 = 0; in2 = 0;  
#100  in1 = 1; in2 = 1;
```

**Results:**

```
At time 0:    out = X  
At time 10:   out -> 1 (Shortest rise delay)  
At time 109:  out -> 0 (Shortest fall delay)
```

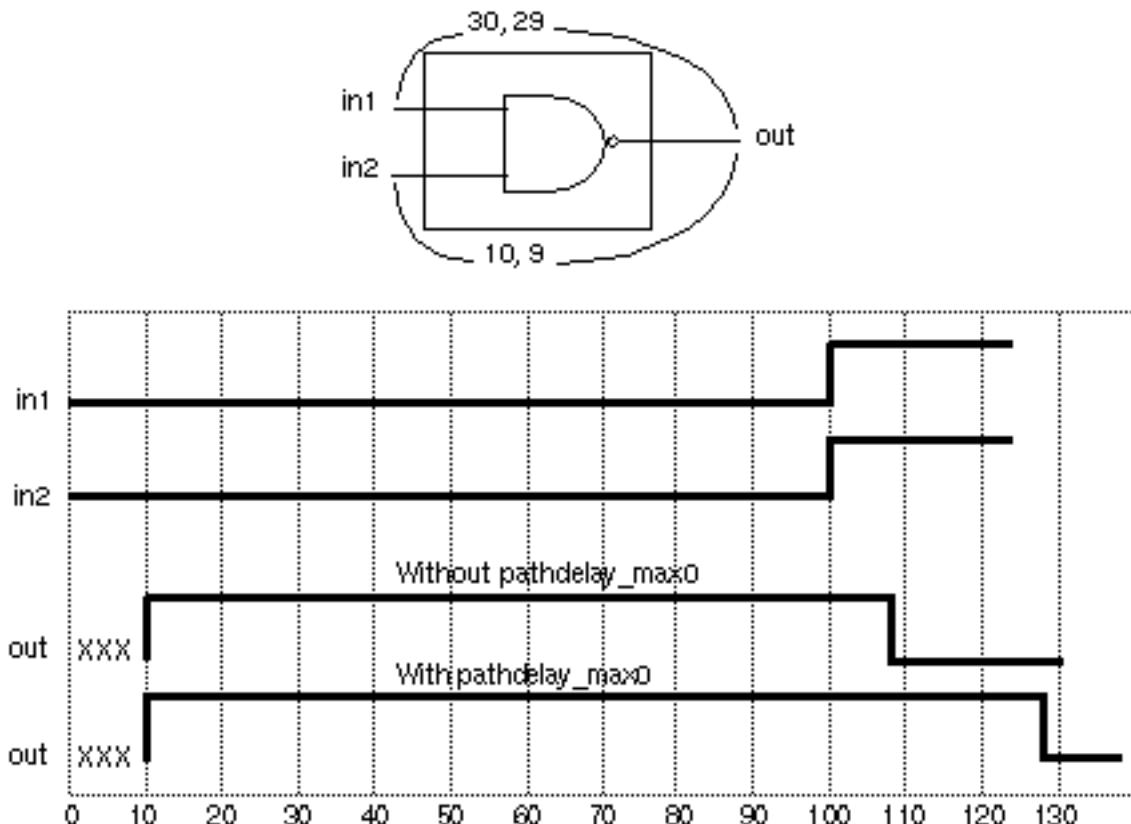
Using the `pathdelay_max0` specify property, as shown in the following specify block, would push the output transition to 0 to time 129 because the longest transition (29) would be selected.

```
specify  
    pathdelay_max0 out;  
    (in1 *> out) = (30, 29);  
    (in2 *> out) = (10, 9);  
endspecify
```

## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

The following figure illustrates the effect of using the `pathdelay_max0` property.



#### Example 2:

Using the same NAND gate with the same rise and fall delays, suppose that the input stimulus is as follows:

Stimulus:

```
#0    in1 = 0; in2 = 0;  
#100  in1 = 1;  
#5    in2 = 1;
```

Results:

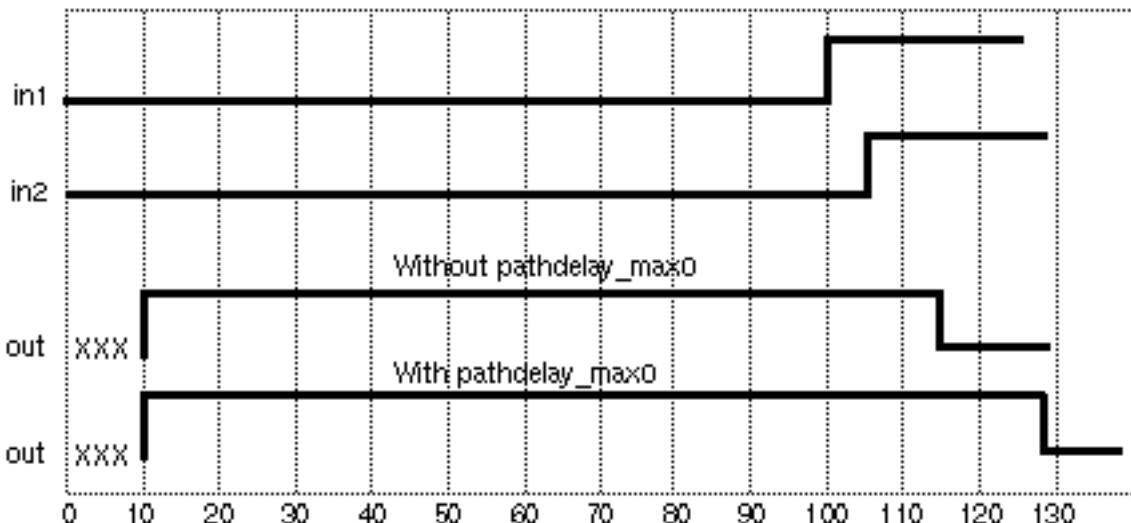
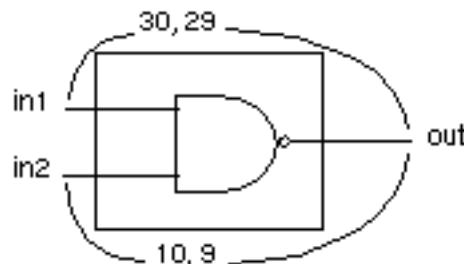
```
At time 0:    out = X  
At time 10:   out -> 1 (Shortest rise delay)  
This change does not affect the output, so  
nothing happens.  
At time 105, out is scheduled to transition  
to 0 at time 114 (105 + 9)  
At time 114, out -> 0
```

By default, the transition of `in2` to 1 at time 105 schedules the output transition to 0 at time 114 (105 + the fall delay for `in2`).

## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

If you use the `pathdelay_max0` property, the longest transition time (the fall delay for `in1`, which is 29) is selected, and `out` transitions to 0 at 129.



#### Example 3:

When `pathdelay_max0` or `pathdelay_max1` is used, an already scheduled output will never be rescheduled for a longer transition time. The calculation is done only at the time when the need to change the output is detected.

The following example uses the same NAND gate and the same rise and fall delays. However, the `pathdelay_max1` directive is used in the specify block.

```
specify
  pathdelay_max1 out;
  (in1 *> out) = (30, 29);
  (in2 *> out) = (10, 9);
endspecify
```

Now suppose that the input stimulus is as follows:

## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

---

#### Stimulus:

```
#0    in1 = 1; in2 = 1;  
  
#100 in2 = 0;  
  
#5    in1 = 0;
```

#### Results:

```
At time 0:   out = X  
At time 9:   out -> 0 (Shortest fall delay)  
Schedule output transition to 1 at time 110  
(rise delay for in2).  
This change does not affect the output, so  
nothing happens.  
At time 110, out -> 1
```

### pathdelay\_controlsignal

The `pathdelay_controlsignal` property specifies that, when an event caused by a control signal is scheduled, all other path delays are to be ignored. The output transition time can be recalculated according to the existing rules being applied to the active path delay(s). This behavior may be desired when modeling something like the enable of a BUFIF gate.

The syntax is as follows:

```
pathdelay_controlsignal output, input [, input ...]
```

Example:

Suppose that you model the rise and fall delays of a BUFIF1 gate as follows:

```
bufif1 (out, in, en);  
specify  
  (in *> out) = (5, 6);  
  (en *> out) = (10, 11, 12);  
endspecify
```

Given the following input stimulus, the results are as follows:

#### Stimulus:

```
#0    in = 0; en = 0;  
#100 en = 1;  
#1    in = 1;
```

#### Results:

```
At time 0: out = X  
At time 6: out -> Z  
At time 106: out -> 1
```

- At time 6, `out` transitions to `Z`. This delay is selected because both `in` and `en` transitioned to 0 at time 0, and the algorithm selects the minimum of `1->Z` (6) and `0->Z` (12).
- At time 100, `en` transitions to 1. This schedules a transition on `out` to 0 at time 111.
- At time 101, `in` transitions to 1. Because this transition schedules a new output value on the output, the transition time on `out` is recalculated. The output transitions from `Z` to 1 at time 106 (101 + 5).

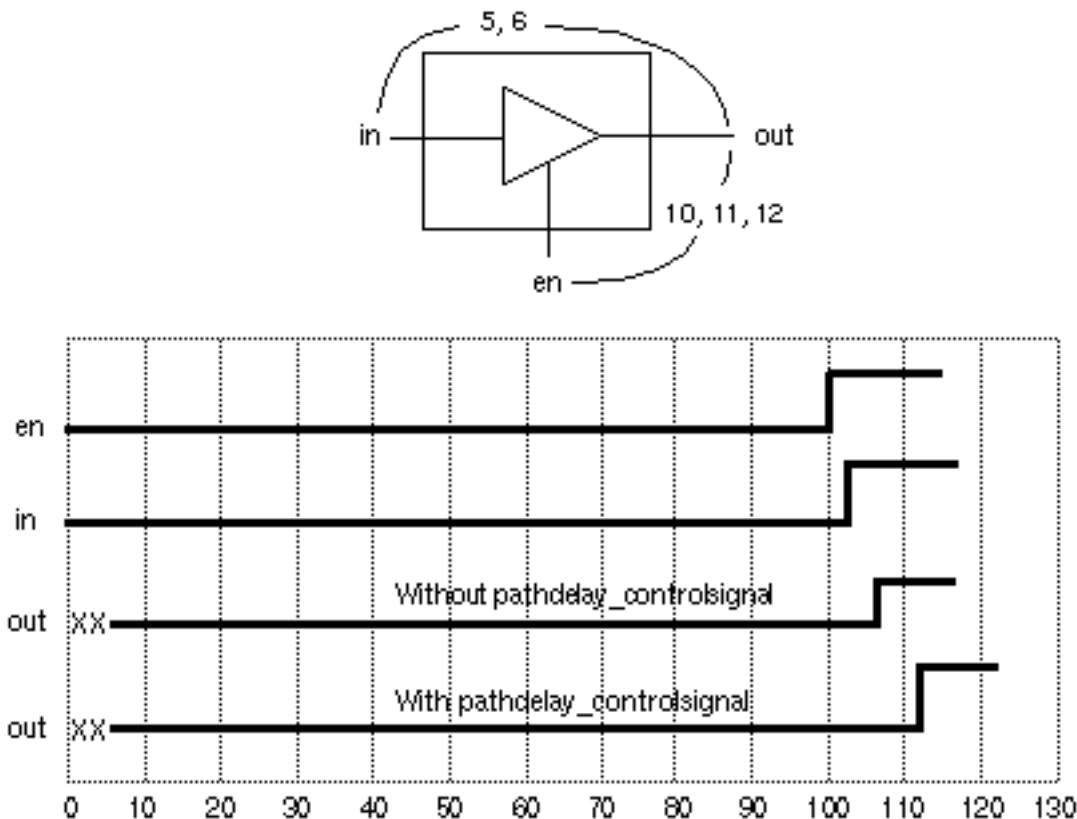
## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

Using the `pathdelay_controlsignal` specify property, as shown in the following specify block, would push the output transition to 1 at time 111 because the transition on `en` from 0 to 1 at time 100 causes the other path delay to be ignored. When `in` transitions to 1 at time 101, the rise delay of the enable signal (10) is used to schedule the output transition.

```
bufif1 (out, in, en);
specify
    pathdelay_controlsignal out, en;
    (in *-> out) = (5, 6);
    (en *-> out) = (10, 11, 12);
endspecify
```

The following figure illustrates the effect of using the `pathdelay_controlsignal` property:

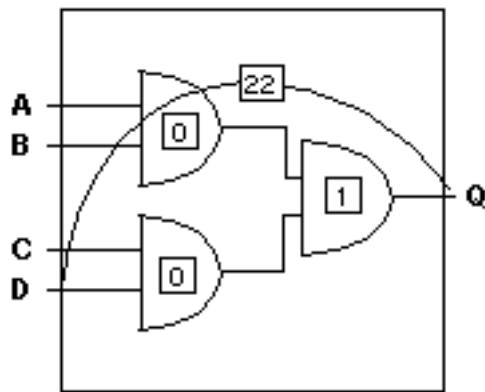


## Mixing Module Path Delays and Distributed Delays

When a module contains both module path delays and distributed delays, the larger of the two delays is used.

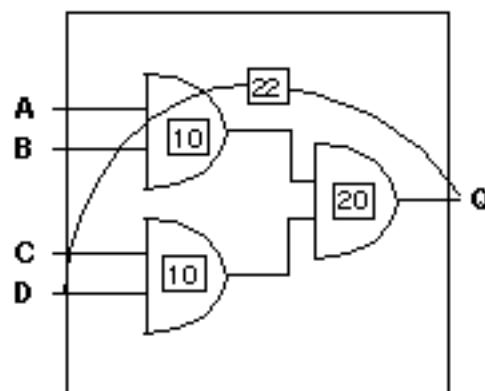
Example 1:

In the following example, 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.



Example 2:

In the following example, 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.

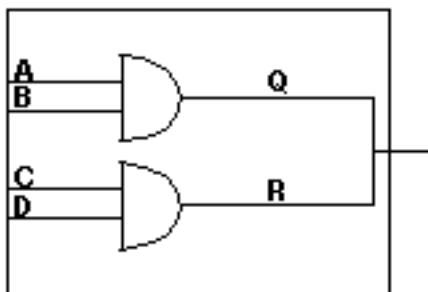
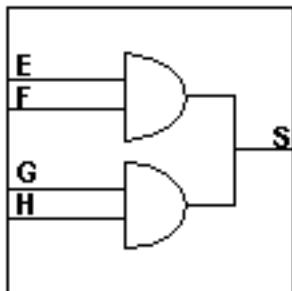


## Strength Changes on Path Inputs

The strength is an implementation function of the internal module. When scheduling module path output events, NC-Verilog 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.

## Driving Wired Logic Outputs

The IEEE 1364 standard (*IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*) states that “Module path output nets shall not have more than one driver within the module. Therefore, wired logic is not allowed at module path outputs”. The specification shows the following two figures as examples of illegal module paths:

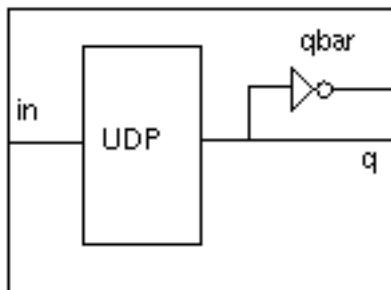


The NC-Verilog simulator, unlike Verilog-XL, does not impose this restriction. You should remember, however, that this is a restriction in the language, and that if you use wired logic at module path outputs, you will not be able to simulate the module path delays with Verilog-XL.

## 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, NC-Verilog will schedule an event on the driven path output later than expected—at the time when the driving path output occurs.

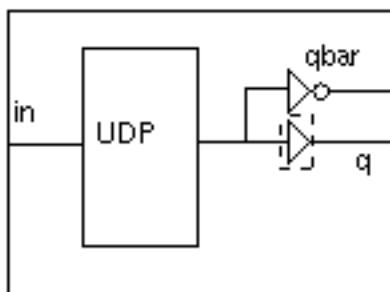
Consider the example in the following figure that shows module path outputs driving other module path outputs.



Pin-to-pin delays:  
 $(in \Rightarrow q) = 12$ ;      DRIVING MODULE PATH  
 $(in \Rightarrow qbar) = 10$ ;      DRIVEN MODULE PATH

In this 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, NC-Verilog 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.

To avoid this situation, place a buffer on the driving output, as shown in the following figure. This creates an internal net to drive  $qbar$  so that any event on  $qbar$  caused by an event on  $in$  will occur after 10 time units.



Pin-to-pin delays:  
 $(in \Rightarrow q) = 12$ ;      DRIVEN MODULE PATH  
 $(in \Rightarrow qbar) = 10$ ;      DRIVEN MODULE PATH

## SDF Annotation of Module Path Delays

You can annotate module paths using an SDF file. Use one of the following SDF file keywords:

- `IOPATH` (see “[IOPATH Keyword](#)” on page 1009 for details)
- `DEVICE` (see “[DEVICE Keyword](#)” on page 1021 for details)

`IOPATH` statements in the SDF file are mapped to corresponding HDL constructs as follows:

- A path with no edges or no conditions in the SDF file, matches any path with the same inputs and outputs.
- A path with an edge in the SDF file must have the same edge in the HDL.
- A path with a condition in the SDF file must have the same condition in the HDL.

The NC-Verilog simulator lets you annotate objects entire module paths or selected sub-paths using the SDF file. For example, assume that you have the following Verilog description:

```
wire [3:0] A, Y;  
specify  
    (A => Y) = (1, 1);  
endspecify
```

An SDF file can contain the following constructs:

```
(IOPATH A Y (3) (4))  
(IOPATH A[3] Y[3] (5) (6))
```

The first statement annotates path `A[3] -> Y[3]`, path `A[2] -> Y[2]`, and so on. The second statement annotates only `A[3] -> Y[3]`. Both entire module path and sub-path annotations can be made to the same module path statement.

**Note:** This is different from Verilog-XL, which requires that you specify whether specify paths are to be expanded or unexpanded using the `+expand_specify_vectors` command line option or the ``expand_specify_vectors` and ``noexpand_specify_vectors` compiler directives.

In the NC-Verilog simulator, only the path specified with the `IOPATH` statement is annotated. For example, assume that you have a module path in a specify block that contains a list of inputs and outputs, such as:

```
(A, B *> Y, Z) = delay
```

## NC-Verilog Simulator Help

### Interconnect and Module Path Delays

---

In Verilog-XL, the following `IOPATH` statement in the SDF file annotates all the paths the source construct represents, while in NC-Verilog, only the sub-path from `A` to `Y` is annotated.

```
(IOPATH A Y (8) (9))
```

The NC-Verilog simulator supports up to 12 delays including pulse limits in the SDF file. Verilog-XL annotation will only annotate up to six delays including pulse limits.

See [Chapter 17, “SDF Timing Annotation,”](#) for more information on SDF annotation.

## SDF Timing Annotation

---

You can annotate the timing check and delay data in an SDF file to Verilog and to VHDL VITAL. SDF annotation is performed during elaboration. The SDF annotator supports SDF versions 1.0, 2.0, 2.1, and 3.0. For versions 2.0 and above, use the `SDFVERSION` statement in the header of the SDF file to specify the version.

This chapter discusses the following topics:

- [VITAL SDF Annotation](#)
- [Verilog SDF Annotation](#)
- [SDF Annotation for Mixed-Language Designs](#)

## VITAL SDF Annotation

In VHDL, you can annotate timing data to VITAL cells only. During annotation, the annotator locates a specified scope and then updates VHDL generics in the cells with the delay and timing constraint data in the SDF file.

To annotate timing data to VITAL cells, you must:

- Compile the SDF file with *ncsdfc*. See “[Compiling the SDF File](#)” on page 790.
- Write an SDF command file. See “[Writing an SDF Command File](#)” on page 791.
- Use the `ncelab -sdf_cmd_file filename` option to include the SDF command file. See “[Specifying an SDF Command File](#)” on page 795.

### Compiling the SDF File

Use the *ncsdfc* utility to compile SDF files. You must compile your SDF files with *ncsdfc* to annotate the timing information that is contained in an SDF file.

This section summarizes how to use *ncsdfc* to compile an SDF file. See “[ncsdfc](#)” on page 906 for full details on using *ncsdfc*.

To compile an SDF file, specify the name of the SDF source file as an argument to the *ncsdfc* command. The syntax is as follows:

```
% ncsdfc [-options] sdf_filename
```

For example, the following command compiles the SDF file called `ibox.sdf`.

```
% ncsdfc -messages ibox.sdf
```

The output of *ncsdfc* is a compiled SDF file called `sdf_filename.x`. For example, if the name of the SDF file is `ibox.sdf`, the output file is called `ibox.sdf.x`. The output file is placed in the current working directory.

You can use the `-output` option to rename the output file. For example, the following command compiles the SDF file called `ibox.sdf`. The `-output` option specifies that the compiled file is to be called `ibox.compiled`.

```
% ncsdfc ibox.sdf -output ibox.compiled
```

You can compress an SDF file before compiling it with *ncsdfc*. For example:

```
% gzip foo.sdf
% ncsdfc foo.sdf.gz
```

The *ncsdfc* command generates `foo.sdf.gz.x`.

## Writing an SDF Command File

An SDF command file contains one or more blocks of statements. There are seven statements. Only one statement is required: the `COMPILED_SDF_FILE` statement, which specifies the compiled SDF file that you want to use. A block can also contain other statements that specify the cell instance to annotate, the name of the log file, scale factors, and so on.

The statements in a command file can be in any order. Use commas to separate the statements, and use a semi-colon after the last statement.

### SDF Command File Statements

- `COMPILED_SDF_FILE = "compiled_sdf_filename"`

Specifies the full or relative path of the compiled SDF file. This statement is required. The argument must be enclosed in quotation marks.

Examples:

```
COMPILED_SDF_FILE = "ipipe.sdf.X"  
  
// SDF file compressed with compress utility and compiled with ncsdfc  
COMPILED_SDF_FILE = "ipipe.sdf.Z.X"  
  
// SDF file compressed with gzip and compiled with ncsdfc  
COMPILED_SDF_FILE = "ipipe.sdf.gz.X"
```

- `SCOPE = instance_path`

Specifies the scope in which the annotation takes place. The annotator uses the hierarchy level of the specified instance to perform the annotation.

This statement is optional. If you do not specify an instance path, the annotator sets the scope to the top level. Any additional instance path information that is contained in the `INSTANCE` statement in the SDF file or in other SDF file constructs, such as `IOPATH`, are added to the path that you specify with `SCOPE`.

Examples:

```
SCOPE = :top  
SCOPE = :top:i1
```

**Note:** You can annotate SDF timing data to Verilog using an SDF command file. For Verilog, the module instance that you specify with the SCOPE statement must use the Verilog hierarchical path syntax. For example, in Verilog, you write the second example above as follows:

```
SCOPE = top.i1
```

■ **CONFIG\_FILE = “*configuration\_filename*”**

Specifies the name of the configuration file. Enclose the name of the configuration file in quotation marks.

This statement is optional. If you do not specify a configuration file, the annotator uses default settings. See [“Using a Configuration File”](#) on page 813 for details on the configuration file.

■ **LOG\_FILE = “*logfile\_name*”**

Specifies the name of the log file. Enclose the name of the log file in quotation marks.

This statement is optional, but the annotator does not generate a log file by default.

Example:

```
LOG_FILE = “sdf.log”
```

■ **MTM\_CONTROL = “*mtm\_spec*”**

Specifies the delay values that you want to annotate. Use one of the following keywords for the *mtm\_spec*:

- MINIMUM—Annotates the minimum delay value.
- TYPICAL—Annotates the typical delay value.
- MAXIMUM—Annotates the maximum delay value.
- TOOL\_CONTROL—Annotates the delay value that is specified on the ncelab command line using the –mindelays, –typdelays, or –maxdelays option. This is the default.

If no command-line option is specified, the default is –typical.

Examples:

```
MTM_CONTROL = “MAXIMUM”
```

```
MTM_CONTROL = “TOOL_CONTROL”
```

■ **SCALE\_FACTORS = “*scale\_factors*”**

Specifies a set of three positive real number multipliers (*min\_mult:typ\_mult:max\_mult*) that the annotator uses to scale the minimum, typical, and maximum timing data from the SDF file before they are annotated.

This statement is optional. If you do not specify values, the default values are 1.0:1.0:1.0 for minimum, typical, and maximum values.

**Example:**

```
SCALE_FACTORS = "1.6:1.4:1.2"
```

■ **SCALE\_TYPE = “*scale\_type*”**

Specifies how the annotator scales the timing specifications in the SDF file. Use one of the following keywords for the *scale\_type*:

- 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**—Scales from the minimum, typical, and maximum timing specifications. This is the default.

**Example:**

```
SCALE_TYPE = "FROM_MINIMUM"
```

## **Example SDF Command Files**

### **Example 1:**

The following SDF command file contains only the required COMPILED\_SDF\_FILE statement to indicate the name of the compiled SDF file.

```
// File dcache.sdf_cmd
COMPILED_SDF_FILE = "dcache.sdf.X";
```

### **Example 2:**

You can annotate a design using multiple SDF files. The following SDF command file contains three COMPILED\_SDF\_FILE statements.

```
// File dcache.sdf_cmd
COMPILED_SDF_FILE = "dcache1.sdf.X";
COMPILED_SDF_FILE = "dcache2.sdf.X";
COMPILED_SDF_FILE = "dcache3.sdf.X";
```

### **Example 3:**

The following SDF command file contains statements that specify the cell to annotate, a log file called `sdf.log`, minimum delay values to be annotated, scale factors for min, typ, and max delay timing specifications, and a scale type that specifies how the annotator scales the timing specifications.

```
// File dcache.sdf_cmd
COMPILED_SDF_FILE = "dcache.sdf.X",
SCOPE = :dcache:i1,
LOG_FILE = "sdf.log",
MTM_CONTROL = "MINIMUM",
SCALE_FACTORS = ".201:1.01:3.01",
SCALE_TYPE = "FROM_MINIMUM";
```

**Note:** If you are using an SDF command file to annotate to Verilog, write the `SCOPE` statement in this example as follows:

```
SCOPE = dcache.i1,
```

### **Example 4:**

The following SDF command file contains separate sections that annotate distinct portions of a design hierarchy.

```
// File sdf.cmd
COMPILED_SDF_FILE = "cpu.sdf.X",
SCOPE = :m1,
LOG_FILE = "cpu_sdf.log";

COMPILED_SDF_FILE = "fpu.sdf.X",
SCOPE = :m2,
LOG_FILE = "fpu_sdf.log";

COMPILED_SDF_FILE = "dma.sdf.X",
SCOPE = :m3,
LOG_FILE = "dma_sdf.log";
```

## Specifying an SDF Command File

After you have compiled your SDF file(s) and written your SDF command file(s), you can run the elaborator with the `-sdf_cmd_file` option to specify the name of the SDF command file(s). For example, in the following command, the `-sdf_cmd_file` option specifies the SDF command file called `dcache.sdf_cmd`.

```
% ncelab -sdf_cmd_file dcache.sdf_cmd top
```

You can repeat the option to specify more than one command file. For example:

```
% ncelab top -sdf_cmd_file cpu.sdf_cmd -sdf_cmd_file ebox.sdf_cmd
```

## Controlling SDF Annotator Output

Error and warning messages that are generated by `ncsdfc` while compiling the SDF file are contained in `ncsdfc.log`. You can suppress the printing of warning messages with the `-neverwarn` option.

The annotator ignores SDF constructs that are not supported by VITAL, and the elaborator issues warning messages. For SDF constructs that are supported, the elaborator generates error messages if it cannot find the corresponding VHDL generics in the model.

The SDF annotator does not generate a log file by default. You must specify the name of the log file by using the `LOG_FILE` statement in the SDF command file. You can use elaborator options such as `-sdf_no_warning` to control the output to the log file.

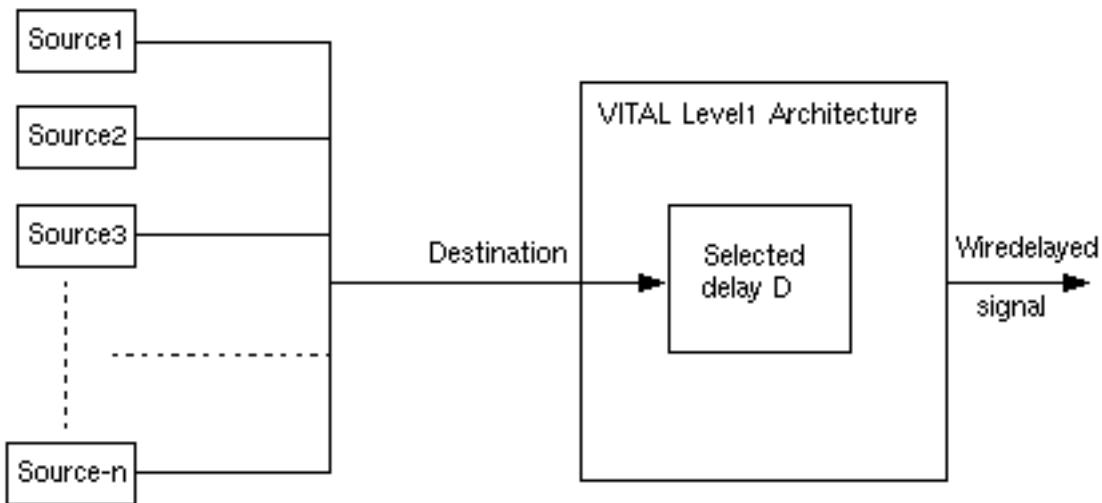
Use the `-sdf_verbose` option if you want to include more detailed information in the log file.

## Multi-Source Interconnect Delays During VITAL SDF Annotation

During VITAL SDF annotation, the SDF annotator can annotate multi-source interconnect delays in two ways:

- Default mode, in which one set of delay values is mapped to the `tipd` generic that is associated with the destination port.

The following figure illustrates the default implementation of multi-source interconnect delays in a VITAL Level1 model.



In this figure, the delay ( $D$ ) is the value of the `tipd` generic that is associated with the destination port. The SDF annotator maps every interconnect construct that has this destination to the `tipd` generic. When more than one SDF construct maps to the same `tipd` generic, the annotator sets the value of the generic to the last interconnect delay that it encounters.

Use the `ncelab -vipdmin` or `-vipdmax` command-line option to select the minimum or maximum of the delay values, respectively.

- Multi-source mode, in which the annotator annotates unique delays for each source-load pair.

Use the `ncelab -intermod_path` command-line option to turn on this functionality.

During elaboration, if the SDF annotator detects that more than one SDF interconnect construct maps to a given `tipd` generic in a VITAL Level1 architecture, the annotator creates separate locations for the delays from each source of the destination port. Each location is initialized to the current value of the `tipd` generic, and then each `INTERCONNECT` entry in the SDF file updates the location corresponding to its source only. Each `PORT` entry updates the locations of all sources of the port.

During simulation, the delay ( $D$  in the figure shown above) is selected dynamically in the following way:

In every simulation cycle in which there is an event on the destination port:

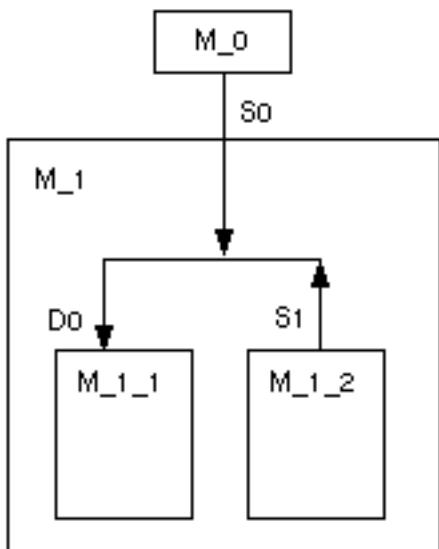
- a. Select all sources of the destination port that have changed in the current simulation cycle.

- b.** Using the new and old values of the destination port, select the appropriate edge-dependent delay for each of the sources that have changed in the current simulation cycle.
- c.** Select the minimum of all the delays identified in step b and set the delay ( $D$ ) to this value.

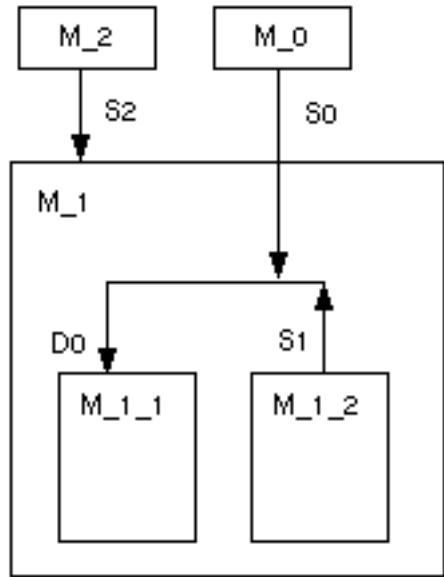
You cannot use the `-vipdmin` or `-vipdmax` options to select the minimum or maximum of the delay values with the `-intermod_path` option.

For a pure VHDL design, programmable pulse limits, if specified, have no effect.

If the source specified in the interconnect specification is not connected directly or indirectly to the destination port in the VHDL model, the SDF annotator ignores the SDF construct and prints a warning message. For example, in the following figure, the two sources  $S_0$  and  $S_1$  are at different levels of hierarchy but are connected to the destination port  $D_0$ . Unique interconnect delay values can be annotated for delays from  $S_0$  and  $S_1$  to  $D_0$ .



In the following figure,  $S_2$  is not connected directly or indirectly to the destination port  $D_0$ . In this case, the annotator will ignore an interconnect specification that has  $S_2$  as the source.



## Command-Line Options that Affect SDF Annotation

The following list shows the elaborator command-line options that affect VITAL SDF annotation.

### **-intermod\_path**

Enable the ability to specify unique delays for multi-source interconnect delays.

### **-maxdelays, -mindelays, -typdelays**

Select the max, min, or typ delay value if a timing triplet in the form  $min:typ:max$  is provided in the SDF file.

### **-noipd**

Ignore the input path delays in a VITAL level 1 cell and directly read the non-delayed input signals.

**-notimingchecks**

Do not execute accelerated VITAL timing checks.

**-no\_sdfa\_header**

Do not print messages that display information that is contained in the SDF command file.

**-no\_tchk\_msg**

Do not display timing check warning messages.

**-no\_tchk\_xgen**

Turn off x-generation in accelerated VITAL timing check procedures.

**-no\_vpd\_msg**

Turn off glitch messages from accelerated VITAL path delay procedures.

**-no\_vpd\_xgen**

Turn off x-generation in accelerated VITAL path delay procedures.

**-ntc\_warn**

Print convergence warnings for negative timing checks if delays cannot be calculated given the current limit values. By default, these warnings are not printed.

**-sdf\_cmd\_file *filename***

Use the specified SDF command file to control SDF annotation.

**-sdf\_no\_warnings**

Do not report warning messages from the SDF annotator.

**-sdf\_verbose**

Include detailed information in the SDF log file.

**-vipdmax**

Select the maximum delay value if more than one interconnect construct in the SDF file maps to the same interconnect path delay generic.

By default, only the last interconnect delay value that the annotator encounters is used as a lumped delay on the destination port. Use the `-vipdmax` option to select the maximum delay value.

Use the `-intermod_path` command-line option if you want to specify unique delay values for multi-source interconnect delays. You cannot use the `-vipdmax` or `-vipdmin` option with the `-intermod_path` option.

**-vipdmin**

Select the minimum delay value if more than one interconnect construct in the SDF file maps to the same interconnect path delay generic.

By default, only the last interconnect delay value that the annotator encounters is used as a lumped delay on the destination port. Use the `-vipdmin` option to select the minimum delay value.

Use the `-intermod_path` command-line option if you want to specify unique delay values for multi-source interconnect delays. You cannot use the `-vipdmax` or `-vipdmin` option with the `-intermod_path` option.

## Verilog SDF Annotation

This section contains the following topics:

- [Overview of Verilog SDF Annotation](#)
- [\\$sdf\\_annotate System Task](#)
- [\\$sdf\\_annotate Examples](#)
- [Requirements for \\$sdf\\_annotate System Tasks](#)
- [Using a Configuration File](#)
- [Using an SDF Command File](#)
- [Controlling SDF Annotator Output](#)
- [Command-Line Options that Affect SDF Annotation](#)

### Overview of Verilog SDF Annotation

SDF annotation is performed during elaboration. The elaborator recognizes `$sdf_annotate` system tasks in your design source files, and if the `$sdf_annotate` system tasks are scheduled to run at time 0, and if they meet other requirements, annotation is performed automatically.

See “[\\$sdf\\_annotate System Task](#)” on page 803 for a description of the `$sdf_annotate` system task. See “[Requirements for \\$sdf\\_annotate System Tasks](#)” on page 811 for a description of the rules that apply to the `$sdf_annotate` tasks for automatic SDF annotation.

The elaborator reads only compiled SDF files. The elaborator automatically calls the `ncsdfc` utility to compile, or recompile, the SDF source file, if necessary.

For the `sdf_file` argument in a `$sdf_annotate` task, you can specify any of the following:

- The name of the SDF source file. For example,

```
$sdf_annotate( "cpu.sdf" );
```

In this case, the elaborator determines that the `$sdf_annotate` argument is a text SDF file, and looks for a corresponding compiled file (`sdf_filelename.X`). For example, if the SDF file is `cpu.sdf`, the elaborator looks for `cpu.sdf.X`.

If the elaborator doesn't find a corresponding compiled file, the elaborator issues a warning message and then spawns the *ncsdfc* utility to automatically compile the SDF file. The compiled SDF file is written to the directory that contains the SDF file.

If the elaborator finds a corresponding compiled file, the elaborator spawns *ncsdfc*, which checks to make sure that the date of the compiled file is newer than the date of the source SDF file and that the version of the compiled file matches the version of *ncsdfc*. If either check fails, *ncsdfc* recompiles the SDF file. Otherwise, the elaborator simply reads the compiled file.

- The name of the compiled SDF file. For example,

```
$sdf_annotation( "cpu.sdf.X" );
```

You can run *ncsdfc* to compile the SDF file and then use the name of the compiled file as the argument to *\$sdf\_annotation*. This lets you use an old compiled file even if a new SDF source file exists. For example, you can run *ncsdfc* to compile *cpu.sdf*, using the *-output* option to generate a compiled file called *cpu\_preroute.sdf*, and then use *cpu\_preroute.sdf* as the argument to *\$sdf\_annotation*. When you elaborate, *ncsdfc* checks that the version of the compiled file matches the version of *ncsdfc*. If the versions match, the elaborator reads the file. If the versions don't match, the *\$sdf\_annotation* task is ignored with a warning.

- The name of a compressed or zipped SDF file. For example,

```
$sdf_annotation( "cpu.sdf.Z" );
$sdf_annotation( "cpu.sdf.gz" );
```

In this case, the elaborator calls *ncsdfc*, which uses *uncompress* or *gzip -d* to read the compressed file and then compiles the SDF file.

- The name of a compressed or zipped and compiled SDF file. For example,

```
$sdf_annotation( "cpu.sdf.Z.X" );
$sdf_annotation( "cpu.sdf.gz.X" );
```

See “[ncsdfc](#)” on page 906 for details on *ncsdfc*.

It is possible to override the default automatic SDF annotation mechanism and force annotation by writing an SDF command file and then including the command file when you elaborate by using the *-sdf\_cmd\_file* option. If you are running the simulator with the *ncverilog* command, include the SDF command file by using the *+sdf\_cmd\_file+* option. See “[Using an SDF Command File](#)” on page 824 for more information.

If you make a change to any SDF-related file (the SDF source file, the compiled SDF file, the SDF configuration file, or the SDF command file), and then execute an *ncsim -update* command, the elaborator automatically re-annotates the design using the new, up-to-date files. See “[Updating Design Changes When You Invoke the Simulator](#)” on page 332 for details on *ncsim -update*.

## \$sdf\_annotate System Task

The syntax of the \$sdf\_annotate system task is as follows:

```
$sdf_annotate ( "sdf_file", [module_instance],  
    ["config_file"], ["log_file"], ["mtm_spec"],  
    ["scale_factors"], ["scale_type"] );
```

The “*sdf\_file*” argument is required. All other arguments are optional. All 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. For example, in the following task, the third (“*config\_file*”) and fourth (“*log\_file*”) arguments are omitted:

```
$sdf_annotate("mysdf.sdf", m1, , , "MAXIMUM", "1:2:3", "FROM_MTM");
```

In the following example, the last three arguments (“*mtm\_spec*”, “*scale\_factors*”, and “*scale\_type*”) are omitted, so the closing parenthesis can follow the last argument present:

```
$sdf_annotate("mysdf.sdf", m1, "mysdf.config", "mysdf.log");
```

The following list describes the \$sdf\_annotate arguments.

### “sdf\_file”

The name of the SDF file. For this argument, you can specify:

- The name of the SDF source file (for example, dcache.sdf).
- The name of the compiled SDF file (for example, dcache.sdf.x).
- The name of a compressed or zipped SDF file (for example, dcache.sdf.gz).
- The name of a compressed or zipped and compiled SDF file (for example, dcache.sdf.gz.x).

See the previous section for details.

### module\_instance

The name of the module instance that you want to annotate.

The instance can have an array index (for example, x.y[3].p) to indicate that the instance is an element in an array of instances.

The SDF annotator uses the hierarchy level of the specified module instance to run the annotation. If you do not specify *module\_instance*, the annotator uses the module that contains the call to the `$sdf_annotate` system task as the *module\_instance* for annotation.

The names in the SDF file are either relative paths to the *module\_instance* or full paths with respect to the entire Verilog HDL description.

#### **“config\_file”**

The name of the configuration file. The configuration file lets you control how the timing data in the SDF file is annotated.

Using a configuration file is optional. The annotator uses default settings if you do not specify a configuration file. See “[Using a Configuration File](#)” on page 813 for a description of the configuration file.

#### **“log\_file”**

The name of the annotation log file. This file contains status information, warnings, and error messages from the SDF annotator. The annotator also prints warning and error messages to standard output.

By default, the annotator does not create an SDF log file. You must include this argument if you want a log file with annotation-specific messages.

#### **“mtm\_spec”**

Specifies the delay values that you want to annotate. *mtm\_spec* is 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—Annotates the delay value that is specified by the command-line option `-mindelays`, `-typdelays`, or `-maxdelays` (`+mindelays`, `+typdelays`, or `+maxdelays` for *ncverilog*).

The default for *mtm\_spec* is TOOL\_CONTROL. If no command-line option is specified, the default is TYPICAL.

**Note:** The *mtm\_spec* argument overrides the `mtm` command in the configuration file.

#### **“scale\_factors”**

Set of three positive real number multipliers that the SDF annotator uses to scale the minimum, typical, and maximum timing values in the SDF file before annotating the values. The syntax of this argument is:

*min\_mult:typ\_mult:max\_mult*

For example:

“1.6:1.4:1.2”

The default for *scale\_factors* is 1.0:1.0:1.0 for minimum, typical, and maximum values.

**Note:** The “*scale\_factors*” argument overrides the `scale` command in the configuration file.

#### **“scale\_type”**

Specifies how the SDF annotator scales the timing specifications. *scale\_type* is 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`—Scales from the minimum, typical, and maximum timing specifications.

The default for *scale\_type* is `FROM_MTM`.

**Note:** The *scale\_type* argument overrides the `scale` command in the configuration file.

### **\$sdf\_annotation Examples**

This section contains five examples of annotation using `$sdf_annotation`.

## Example 1

In the following example, timing information in a file called `my.sdf` is used to annotate module instance `top.m1`.

```
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
```

In the `$sdf_annotate` system task shown in this example:

- "`my.sdf`" is the name of the SDF file.
- `m1` is the module instance that the `$sdf_annotate` task annotates.
- The third argument, the configuration file name, is omitted, as is the fourth argument, the name of the log file. By default, no SDF log file is generated. The commas separating these omitted arguments are required.
- The "`mtm_spec`" argument "`MAXIMUM`" specifies that the maximum delays in the SDF file are annotated to the design.
- The fifth argument specifies scale factors of `1.6`, `1.4`, and `1.2`. These scale factors multiply the members of all delay triplets in the SDF file to create new triplets, whose members are then annotated to the design.
- The last argument specifies that the scale type is `FROM_MTM`, which specifies that scale factor `1.6` multiplies the minimum members in the SDF file delay triplets, `1.4` multiplies the typical members, and `1.2` multiplies the maximum members.

In this example, a timing triplet in the SDF file of `1.0:2.0:3.0` is changed to `1.6:2.8:3.6`. This is arrived at by:

- Multiplying the min value in the SDF file (`1.0`) by scale factor `1.6`
- Multiplying the typ value in the SDF file (`2.0`) by scale factor `1.4`
- Multiplying the max value in the SDF file (`3.0`) by scale factor `1.2`

The delay in the maximum position in the new triplet (`3.6`) is then used for annotation.

## Example 2

The following example is identical to Example 1, except that the “*scale\_type*” argument has been changed to “FROM\_MAXIMUM”.

```
module top;
    ...
    circuit m1(i1,i2,i3,o1,o2,o3);
    initial
        $sdf_annotate("my.sdf", m1, , , "MAXIMUM", "1.6:1.4:1.2", "FROM_MAXIMUM");
        //stimulus and response checking
    ...
    ...
endmodule
```

In this example, a timing triplet in the SDF file of  $1.0:2.0:3.0$  is changed to  $4.8:4.2:3.6$ . This is arrived at by:

- Multiplying the max value in the SDF file (3.0) by scale factor 1.6
- Multiplying the max value in the SDF file (3.0) by scale factor 1.4
- Multiplying the max value in the SDF file (3.0) by scale factor 1.2

The delay in the maximum position in the new triplet (3.6) is then used for annotation.

## Example 3

In the following example, `$test$plusargs` is used to check for the presence of plus options on the command line. If `+preroute` appears on the command line, the file `preroute.sdf` is used for annotation. If `+postroute` appears on the command line, the file `postroute.sdf` is used.

```
module top;
    ...
    circuit m1(i1,i2,i3,o1,o2,o3);
    initial
        if ($test$plusargs("preroute"))
            $sdf_annotate("preroute.sdf", m1);
        else
            if ($test$plusargs("postroute"))
                $sdf_annotate("postroute.sdf", m1);
            else
                $display("No SDF annotation being done for this run");
        //stimulus and response checking

```

```
...  
endmodule
```

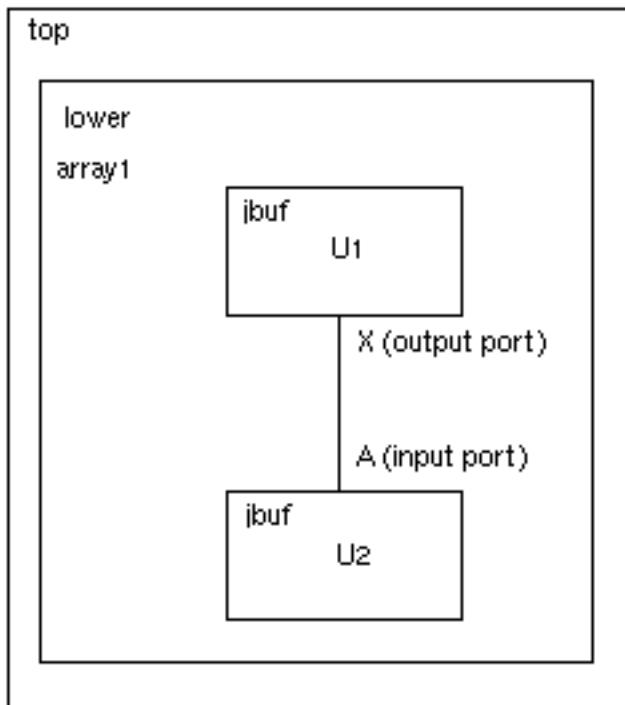
#### **Example 4**

The following example shows separate annotations to distinct portions of a design hierarchy. 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);  
  
    // 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
```

## Example 5

The following example illustrates the annotation of an interconnect delay that occurs between the output of one instance (instance u1 of jbuf) and the input of another instance (instance u2 of jbuf). The following figure illustrates the hierarchy.



The HDL code for the example is shown below. Notice that the only argument to the \$sdf\_annotation task is the name of the SDF file.

```
'timescale 1ns/1ns
module top();
    reg t;
    lower array1(f,t);
    initial
        begin
            $sdf_annotation("top.sdf");
        end
    initial
        begin
            fork
                #00 t = 0;
                #20 t = 1;
                #60 t = 0;
            end
        end
```

## NC-Verilog Simulator Help

### SDF Timing Annotation

---

```
#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
```

The SDF file is shown below:

```
(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
```

The delay between the instance ports is indicated in this SDF file by the `INTERCONNECT` keyword followed by port descriptions. The first port is an output or inout, and the second port is an input or inout.

In this example, the full hierarchical description of the ports is given in the SDF file. The lowest level in the description of each port 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_annotation` task and the SDF file, with the information in the task constituting the beginning of the information. You can include or omit the level containing the task (`top`, in this example).

The following table shows the configurations of hierarchical information that you can use in this example.

<b><code>\$sdf_annotation module_instance argument</code></b>	<b>Interconnect ports in SDF file</b>
No entry (as in the current example)	<code>array1.u1.X array1.u2.A</code>
<code>top</code>	<code>array1.u1.X array1.u2.A</code>
<code>top.array1</code>	<code>u1.X u2.A</code>
<code>array1</code>	<code>u1.X u2.A</code>

## Requirements for `$sdf_annotation` System Tasks

The elaborator ignores and generates a warning for any `$sdf_annotation` system task that does not satisfy the following rules:

- `$sdf_annotation` tasks must be inside an `initial` block. For example,

```
initial  
  $sdf_annotation("../DataFiles/dramctrl.sdf",test.top,,, "MAXIMUM");
```

A `$sdf_annotation` task cannot be referenced in a task call contained in an `initial` block. The following code results in an error because the `$sdf_annotation` task is not inside an `initial` block:

```
initial
begin
    tasksdf;
end
...
...
...
task tasksdf;
begin
    $sdf_annotate("../DataFiles/dramctrl.sdf", test.top, , "MAXIMUM");
end
endtask
```

- Only `$sdf_annotate` tasks scheduled to run at time 0 are used for annotation.
- Delay or event control statements cannot precede `$sdf_annotate` calls. For example, all of the following calls are ignored with a warning:

```
initial
begin
    #0 in = 0;
    $sdf_annotate(args...);
end

initial
begin
    #10 out = 56;
    $sdf_annotate(args...);
end

initial
begin
    @posedge(clk)
    $sdf_annotate(args...);
end
```

- `$sdf_annotate` calls cannot be within or follow `for`, `while`, `case`, `repeat`, or `wait` constructs.
- Because annotation takes place at elaboration time, and the values of variables in the design are determined at simulation time, a `$sdf_annotate` task cannot be invoked from an `if` construct with a variable expression as the condition. The expression that is used in the guard expression must evaluate to a constant. For example, if `count` is a net or register in the design, the following call results in an error that causes the elaborator to exit:

```
initial
  if (count == 1)
    $sdf_annotate(args...);
```

If a `$sdf_annotate` task violates the above requirements, the elaborator generates warning messages telling you that it is ignoring the system task. The following example illustrates the warning messages that are generated if a `$sdf_annotate` task is preceded by a delay statement.

```
ncelab: *W,CUSDEC: A Delay or an Event Control was found before the SDF System Task..
$sdf_annotate("my.sdf", , , "sdf1.log");
|
ncelab: *W,CUSSTI (.top.v,12|12): This SDF System Task will be Ignored..
$sdf_annotate("my.sdf", , , "sdf1.log");
```

## Using a Configuration File

The configuration file lets you control how the timing data in the SDF file is annotated. Using a configuration file is optional. The annotator uses default settings if you do not specify a configuration file in the `$sdf_annotate` task or in the SDF command file.

You can do the following by using a configuration file:

- Ignore the timing constructs in the SDF file.
- Select the minimum, typical, or maximum delay values.
- Specify scaling operations
- Determine turn-off delays
- Map annotation data when the SDF file and the module differ

**Note:** The NC-Verilog simulator annotates the design even if the annotator finds a problem in the configuration file.

## Configuration File Syntax

This section shows the syntax of the configuration file. All keywords must be in uppercase, and no blank lines are allowed.

**Note:** In the following syntax description, square brackets enclosing a list of arguments separated by OR bars indicates that you must select one of the enclosed arguments.

## NC-Verilog Simulator Help

### SDF Timing Annotation

---

```
sdf_construct = IGNORE;

INTERCONNECT_DELAY = [MINIMUM | MAXIMUM | AVERAGE];

INTERCONNECT_MIPD = [MINIMUM | MAXIMUM | AVERAGE];

MTM = [MINIMUM | TYPICAL | MAXIMUM | TOOL_CONTROL];

SCALE_FACTORS = min_mult:typ_mult:max_mult;

SCALE_TYPE = [FROM_MINIMUM | FROM_TYPICAL | FROM_MAXIMUM | FROM_MTM];

TURNOFF_DELAY = [MINIMUM | MAXIMUM | AVERAGE | FROM_FILE];

MODULE module_name
{
    MTM = [MINIMUM | TYPICAL | MAXIMUM];
    SCALE_FACTORS = min_mult:typ_mult:max_mult;
    SCALE_TYPE = [FROM_MINIMUM | FROM_TYPICAL | FROM_MAXIMUM | FROM_MTM];

    MAP_INNER = hierarchical_path;
    [ (original_timing) = ADD {(new_timing); ...; }
    | (original_timing) = OVERRIDE {(new_timing); ...; }
    | (original_timing) = IGNORE;
}

}
```

## Example Configuration File

All of the configuration file keywords are shown in the following example. If the SDF annotator finds conflicting keywords, it uses the last specified keyword.

```
PATHPULSE = IGNORE;           // Ignore all PATHPULSE constructs in SDF file.
INTERCONNECT_DELAY = MAXIMUM; // Use maximum interconnect delay.
MTM = MAXIMUM;               // Use maximum delays from SDF.
SCALE_FACTORS = 0.5:1:2.0;    // Scale the delays with these factors.
SCALE_TYPE = FROM_TYPICAL;   // Scale from the typical delays.
TURNOFF_DELAY = FROM_FILE;   // Use the turn-off delays in SDF.
MODULE AND                  // Applies to instances of type AND.
{
    MAP_INNER = and1;         // Map delays to inner module and1.
    (in1 => out1) = OVERRIDE // Use delays specified between in1 and
                           // out1 in the SDF file to override the
```

## NC-Verilog Simulator Help

### SDF Timing Annotation

---

```
(CP => Q);          // delay paths between CP and Q specified
}                   // in the module instance and1.
}
```

### IGNORE Keyword

The IGNORE keyword specifies that the annotator should ignore the timing check or delay construct in the SDF file. See [Appendix B, “SDF File Syntax,”](#) for details on SDF file keywords.

```
DEVICE = IGNORE;
HOLD = IGNORE;
INTERCONNECT = IGNORE;
IOPATH = IGNORE;
NETDELAY = IGNORE;
NOCHANGE = IGNORE;
PATHPULSE = IGNORE;
PATHPULSEPERCENT = IGNORE;
PERIOD = IGNORE;
PORT = IGNORE;
RECOVERY = IGNORE;
SETUP = IGNORE;
SETUPHOLD = IGNORE;
SKEW = IGNORE;
WIDTH = IGNORE;
```

### Default Mapping for the NC-Verilog simulator

If you do not specify that a timing construct is to be ignored, the SDF annotator uses the default mapping for that keyword as shown in the following table.

---

SDF Timing Keywords	Path delay library	Distributed delay library
DEVICE	PATH	LUMPED OUTPUT
HOLD	HOLD	
INTERCONNECT	INTERCONNECT DELAY	INTERCONNECT DELAY
IOPATH	PATH	LUMPED OUTPUT
NETDELAY	INTERCONNECT DELAY	INTERCONNECT DELAY
PERIOD	PERIOD	

## NC-Verilog Simulator Help

### SDF Timing Annotation

---

SDF Timing Keywords	Path delay library	Distributed delay library
PORT	INTERCONNECT DELAY	INTERCONNECT DELAY
RECOVERY	RECOVERY	
SETUP	SETUP	
SETUPHOLD	SETUP/HOLD	
SKEW	SKEW	
WIDTH	WIDTH	

#### **INTERCONNECT\_DELAY Keyword**

**Note:** The INTERCONNECT\_DELAY construct is not currently implemented in the NC-Verilog simulator.

The INTERCONNECT\_DELAY keyword specifies which interconnect delay to use when there are multiple annotations to the same source/load pair. You can select one of the following arguments:

```
INTERCONNECT_DELAY = [MINIMUM | MAXIMUM | AVERAGE];
```

---

Keyword	Description
MINIMUM	Annotates the minimum delay.
MAXIMUM	Annotates the maximum delay.
AVERAGE	Annotates the average of all the delays.

---

Example:

If you have multiple annotations to the same source/load pair, the annotator uses the delays in the last INTERCONNECT statement. For example, if you have the following two statements, the annotator uses the second one.

```
(INTERCONNECT out1 in3 (5) (6))  
(INTERCONNECT out1 in3 (7) (8))
```

If you want to annotate the minimum delays, use the following configuration file statement:

```
INTERCONNECT_DELAY = MINIMUM;
```

## **INTERCONNECT\_MIPD Keyword**

**Note:** The `INTERCONNECT_MIPD` construct is not currently implemented in the NC-Verilog simulator.

The `INTERCONNECT_MIPD` construct performs the same function as the `INTERCONNECT_DELAY` construct. It is provided for compatibility with Verilog-XL.

## **MTM Keyword**

The `MTM` keyword specifies whether the SDF annotator uses the minimum, typical, or maximum delays from the SDF file. The syntax for the `MTM` keyword is as follows:

```
MTM = [MINIMUM | TYPICAL | MAXIMUM | TOOL_CONTROL];
```

---

<b>Keyword</b>	<b>Description</b>
MINIMUM	Annotates the minimum delay value.
TYPICAL	Annotates the typical delay value. This is the default.
MAXIMUM	Annotates the maximum delay value.
TOOL_CONTROL	Annotates the delay value specified by the option that you use when you invoke the elaborator: <code>-mindelays</code> , <code>-typdelays</code> , or <code>-maxdelays</code> .

---

**Note:** The `MTM_CONTROL` statement in the SDF command file overrides the `MTM` keyword in the configuration file.

## **SCALE\_FACTORS Keyword**

The `SCALE_FACTORS` keyword specifies the scaling operations that the SDF annotator performs on the timing information before it is annotated. The syntax for the `SCALE_FACTORS` keyword is as follows:

```
SCALE_FACTORS = min_mult:typ_mult:max_mult;
```

The argument to the `SCALE_FACTORS` keyword is a set of three positive real number multipliers. If you do not specify values, the default values are `1.0:1.0:1.0` for the minimum, typical, and maximum values.

**Note:** The `SCALE_FACTORS` statement in the SDF command file overrides the `SCALE_FACTORS` keyword in the configuration file.

Example:

```
SCALE_FACTORS = 1.6:1.4:1.2;
```

### **SCALE\_TYPE Keyword**

The SCALE\_TYPE keyword specifies the scale type that the annotator uses when it performs scaling operations on the timing information before it is annotated. The syntax for the SCALE\_TYPE keyword is as follows:

```
SCALE_TYPE = [ FROM_MINIMUM | FROM_TYPICAL | FROM_MAXIMUM | FROM_MTM ] ;
```

---

<b>Keyword</b>	<b>Description</b>
FROM_MINIMUM	Scales from the minimum timing specification in the SDF file.
FROM_TYPICAL	Scales from the typical timing specification in the SDF file.
FROM_MAXIMUM	Scales from the maximum timing specification in the SDF file.
FROM_MTM	Scales directly from the minimum, typical, or maximum timing specifications in the SDF file, as selected by <a href="#">MTM</a> in the configuration file, <a href="#">MTM CONTROL</a> in the SDF command file, and command-line options. This is the default.

---

**Note:** The [SCALE\\_TYPE](#) statement in the SDF command file overrides the SCALE\_TYPE keyword in the configuration file.

Example:

The following example shows how the SCALE\_FACTORS and SCALE\_TYPE keywords in the configuration file affect the timing specifications in the SDF file. In the following CELL entry, the NETDELAY keyword is used to assign rise, fall, and turn-off delays to a net in the cell instance x.

```
(CELL (CELLTYPE "adder4")
  (INSTANCE x)
  (DELAY
    (ABSOLUTE (NETDELAY a.o2 (6:7:8) (4:6:7) (5:8:9)))
  )
)
```

The configuration file defines the scale factors and scale type as follows:

```
SCALE_FACTORS = 0.5:1:1.5;  
SCALE_TYPE = FROM_TYPICAL;  
MTM = MINIMUM;
```

The typical delays in the SDF file are multiplied by the minimum scale factor to annotate the following rise, fall, and turn-off delays for the net:

```
(3.5) (3) (4)
```

### **TURNOFF\_DELAY Keyword**

**Note:** The TURNOFF\_DELAY construct is not currently implemented in the NC-Verilog simulator.

The TURNOFF\_DELAY keyword specifies how the SDF annotator determines the turn-off delay that is annotated. The syntax for the TURNOFF\_DELAY keyword is as follows:

```
TURNOFF_DELAY=[MINIMUM | MAXIMUM | AVERAGE | FROM_FILE];
```

---

<b>Keyword</b>	<b>Description</b>
MINIMUM	Choose the smallest values from the rise and fall delays. This is the default.
MAXIMUM	Choose the greatest values from the rise and fall delays.
AVERAGE	Average the values from the rise and fall delays.
FROM_FILE	Use the turn-off delays in the SDF file. If you do not specify FROM_FILE, or if you specify FROM_FILE but the SDF file does not contain the turn-off delay, the turn-off delay is set to MINIMUM(rise, fall).

---

Example:

The following CELL entry in the SDF file assigns rise, fall, and turn-off delays to a net in the cell instance x.

```
(CELL (CELLTYPE "adder4")  
(INSTANCE x)  
(DELAY  
  (ABSOLUTE (NETDELAY a.o2 (6:7:8) (4:6:9) (5:8:10))))
```

If the configuration file sets the TURNOFF\_DELAY keyword to MAXIMUM, then the turn-off delay for annotation is ( 6 : 7 : 9 ), which is derived from the maximum of the rise and fall min:typ:max values:

```
maximum (6, 4):maximum (7, 6):maximum (8, 9)
```

## MODULE Keyword

The MODULE keyword maps path delays and timing checks from the SDF file to the Verilog HDL description, performs scaling operations for a specific type of module, and selects min:typ:max delay data.

The syntax for the MODULE keyword is as follows:

```
MODULE module_name
{
    MTM = [MINIMUM | TYPICAL | MAXIMUM];
    SCALE_FACTORS = min_mult:typ_mult:max_mult;
    SCALE_TYPE = [FROM_MINIMUM | FROM_TYPICAL | FROM_MAXIMUM | FROM_MTM];
    MAP_INNER = hierarchical_path;
    [ (original_timing) = ADD {(new_timing); ...; }
     | (original_timing) = OVERRIDE {(new_timing); ...; }
     | (original_timing) = IGNORE; ]
}
```

**Note:** The MTM, SCALE\_FACTORS, and SCALE\_TYPE arguments to the MODULE keyword affect only the IOPATH, DEVICE, and TIMINGCHECK information that is annotated to the specified module.

---

Argument	Description
<i>module_name</i>	Name of a specific type of module (not instance name) specified in the Verilog HDL description.
MTM	See the “ <a href="#">MTM Keyword</a> ” on page 817 for details.
SCALE_FACTORS	See the “ <a href="#">SCALE FACTORS Keyword</a> ” on page 817 for details.
SCALE_TYPE	See the “ <a href="#">SCALE TYPE Keyword</a> ” on page 818 for details.
MAP_INNER	See the “ <a href="#">MAP INNER Keyword</a> ” on page 821 for details.

---

## MAP\_INNER Keyword

The MAP\_INNER keyword specifies a submodule in the hierarchy of the module specified with the MODULE keyword. You can use MAP\_INNER for any number of submodules contained within the module.

The syntax for the MAP\_INNER keyword is as follows:

```
MAP_INNER = hierarchical_path;  
[ (original_timing) = ADD {(new_timing); ...; }  
| (original_timing) = OVERRIDE {(new_timing); ...; }  
| (original_timing) = IGNORE; ]
```

---

Argument	Description
<i>hierarchical_path</i>	Verilog HDL hierarchical path of a submodule of the module specified with the MODULE keyword. The paths specified in the SDF file are mapped to this instance within the module. This path applies to all path delays and timing checks specified for this module in the SDF file and that are mentioned in ADD, OVERRIDE, or IGNORE statements, up until the next MAP_INNER statement.
<i>original_timing</i>	Verilog HDL syntax of the path delay or timing check from the SDF file.
<i>new_timing</i>	Verilog HDL syntax of the path delay or timing check to which to map.
ADD	Annotate not only to the original timing construct in the mapped instance, but also to these additional constructs.
OVERRIDE	Annotate to the provided list of constructs instead of to the construct as specified in the SDF file.
IGNORE	Ignore the matching timing specifications in the SDF file.

---

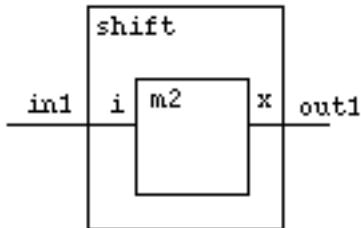
**Note:** In all cases, the *hierarchical\_path* name is applied to all *new\_timing* specifications before they are annotated. In the case of ADD, the *hierarchical\_path* name is applied to the original timing specification.

## NC-Verilog Simulator Help

### SDF Timing Annotation

#### Example 1:

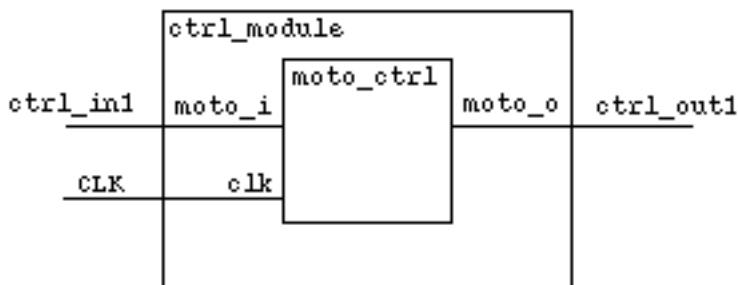
This example applies module mapping to the `shift` module type, which contains a submodule called `m2`. For this module, the minimum delays in the SDF file are to be used, and the scale factor is 2.0. For submodule `m2`, the delay between `in1` and `out1` in the SDF file is to be mapped to the delay between `i` and `x`.



```
MODULE shift
{
    MTM = MINIMUM;
    SCALE_FACTORS = 2.0:2.0:2.0;
    MAP_INNER = m2;
    (in1 => out1) = OVERRIDE
    {
        (i => x); }
}
```

#### Example 2:

In this example, two mappings are performed with the `MAP_INNER` keyword. Using the `OVERRIDE` keyword with the hierarchical design in the following figure, this example shows you how to map and annotate the delay from the path `ctrl_in1=>ctrl_out1` to the path `moto_i=>moto_o`, and `clk=>moto_o`.



The Verilog design has a specify block with the following path delay specification:

```
(moto_i => moto_o) = (3, 4);
```

## NC-Verilog Simulator Help

### SDF Timing Annotation

---

The SDF file for the design has the following delay specification for the path:

```
(IOPATH ctrl_in1 ctrl_out1 (5) (6))
```

A module mapping for this hierarchy is specified in the SDF configuration file as follows:

```
MODULE ctrl_module
{
    MAP_INNER = moto_ctrl;
    (ctrl_in1 => ctrl_out1) = OVERRIDE
    {
        (moto_i => moto_o);
        (clk => moto_o); }
}
```

#### **Example 3:**

Using the IGNORE keyword with the same hierarchy as shown in Example 2, the following mapping in the configuration file ignores the specification in the SDF file and continues to use the timing in the Verilog design.

```
MODULE ctrl_module
{
    MAP_INNER = moto_ctrl;
    (ctrl_in1 => ctrl_out1) = IGNORE;
}
```

#### **Rules for Module Mapping with Conditional Delays**

The rules for module mapping in the case of conditional path delays are shown in the following tables. The first table shows the different ways in which Verilog design path delays are handled in the SDF annotation process, when combined with the COND statements in the SDF file. The second table shows the rules for mapping paths between the SDF file and the configuration file.

#### **Annotating Path Delays in the NC-Verilog simulator**

SDF	NC-Verilog	Annotate action
no condition	no condition	Annotate one path
no condition	conditional path delay	Annotate to all conditions in the design, unless an ifnone is present
COND	no condition	No annotation

SDF	NC-Verilog	Annotate action
COND	conditional path delay	Annotate one path

## Module Mapping in SDF

SDF File	Config File	Map action
no condition	no condition	Mapping performed
no condition	COND	No mapping performed
COND	no condition	Mapping performed
COND	COND	Mapping performed

## Using an SDF Command File

If the `$sdf_annotate` system tasks in the design do not meet the requirements listed in “[Requirements for \\$sdf\\_annotate System Tasks](#)” on page 811, you can override the automatic annotation mechanism and force the annotation. To do this:

- Compile the SDF file with `ncsdfc`.

To compile an SDF file, specify the name of the SDF source file as an argument to the `ncsdfc` command. The syntax is as follows:

```
% ncsdfc [-options] sdf_filename
```

For example, the following command compiles the SDF file called `ibox.sdf`.

```
% ncsdfc -messages ibox.sdf
```

The output of `ncsdfc` is a compiled SDF file called `sdf_filename.x`. For example, if the name of the SDF file is `ibox.sdf`, the output file is called `ibox.sdf.x`. The output file is placed in the current working directory.

You can use the `-output` option to rename the output file. For example, the following command compiles the SDF file called `ibox.sdf`. The `-output` option specifies that the compiled file is to be called `ibox.compiled`.

```
% ncsdfc ibox.sdf -output ibox.compiled
```

You can compress an SDF file before compiling it with `ncsdfc`. For example:

```
% gzip foo.sdf
% ncsdfc foo.sdf.gz
```

The `ncsdfc` command generates `foo.sdf.gz.x`.

See “[ncsdfc](#)” on page 906 for more details on compiling an SDF file with `ncsdfc`.

- Write an SDF command file.

An SDF command file contains one or more blocks of statements. There are seven statements, which correspond to the seven arguments of the `$sdf_annotate` system task. Only one statement is required: the `COMPILED_SDF_FILE` statement, which specifies the compiled SDF file that you want to use.

The statements in a command file can be in any order. Use commas to separate the statements, and use a semi-colon after the last statement.

An SDF command file is required if you are annotating to VHDL VITAL cells, and the syntax of the command file is described in the section on VITAL SDF annotation. See “[Writing an SDF Command File](#)” on page 791 for details on the SDF command file and for examples.

If you have already run `ncelab`, or have run `ncverilog`, the elaborator generates warning messages if the `$sdf_annotate` system tasks in the design do not meet the requirements for automatic SDF annotation. These messages include a warning message similar to the following example. You can use the information in the warning message to create an SDF command file.

```
ncelab: *W,CUSDFI (./top.v,11|12): To force annotation, use option  
-SDF_CMD_FILE <cmd_file_name>.// SDF Command file// Cut and Paste the next set of lines into a SDF command file if it is necessary to force annotation. Please remember to compile my.sdf before using it.COMPILED_SDF_FILE = "my.sdf.X",SCOPE = top,LOG_FILE = "sdf1.log";
```

**Note:** At the time when the elaborator generates these warning messages, the scope argument to the `$sdf_annotate` task (the second argument) cannot be fully resolved. After cutting and pasting the line(s) in the warning message into an SDF command file, check the command file and modify any `SCOPE` statements, if necessary, so that the proper scope is specified for annotation.

- Use the `ncelab -sdf_cmd_file filename` option (+`sdf_cmd_file+filename` option if you are using `ncverilog`) to include the SDF command file.

For example, in the following command, the `-sdf_cmd_file` option specifies the SDF command file called `dcache.sdf_cmd`.

```
% ncelab -sdf_cmd_file dcache.sdf_cmd top
```

You can repeat the option to specify more than one command file. For example:

```
% ncelab top -sdf_cmd_file cpu.sdf_cmd -sdf_cmd_file ebox.sdf_cmd
```

## Controlling SDF Annotator Output

Log and error messages generated by the elaborator and by *ncsdfc* while compiling the SDF file specified in a `$(sdf_annotate())` system task are contained in `ncelab.log`. If you run the NC-Verilog simulator using the `ncverilog` command, these messages are contained in `ncverilog.log`.

The SDF annotator does not generate a log file by default. You must specify the name of the log file either by using the `log_file` argument of the `$(sdf_annotate)` system task or by using the `LOG_FILE` statement in an SDF command file.

You can use elaborator options such as `-sdf_no_warning` to control the output to the log file. If you run the NC-Verilog simulator using the `ncverilog` command, the Verilog-XL command-line options that you use to suppress warning or error messages, such as `+nosdfwarn`, `+sdf_no_errors`, and `+sdf_no_warnings`, are translated to the corresponding NC-Verilog elaborator command-line options.

Use the `-sdf_verbose` option if you want more detailed information included in the log file. If you run the NC-Verilog simulator using the `ncverilog` command, the Verilog-XL `+sdf_verbose` command-line option is translated to `ncelab -sdf_verbose`.

## Command-Line Options that Affect SDF Annotation

The following tables list the elaborator and simulator command-line options that have an effect on SDF annotation. The second column shows the equivalent Verilog-XL options.

---

<b>Elaborator (ncelab) options:</b>	<b>Verilog-XL</b>
<code>-epulse_neg</code>	<code>+show_cancelled_e</code>
Filter cancelled events (negative pulses) to the <code>e</code> state. This option makes cancelled events visible.	
<code>-epulse_noneg</code>	<code>+no_show_cancelled_e</code>
Do not filter cancelled events (negative pulses) to the <code>e</code> state.	

## NC-Verilog Simulator Help

### SDF Timing Annotation

---

<b>Elaborator (ncelab) options:</b>	<b>Verilog-XL</b>
-epulse_ondetect	+pulse_e_style_ondetect
Use On-Detect filtering of error pulses. This option extends the e state back to the edge of the event that caused the pulse to occur.	
-epulse_onevent	+pulse_e_style_onevent
Use On-Event filtering of error pulses.	
-intermod_path	Two command-line options: +multisource_int_delays and +transport_int_delays
Enable transport delay behavior with pulse control and the ability to specify unique delays for each source-load path.	
-maxdelays	+maxdelays
Apply the maximum delay value if a timing triplet in the form <i>min:typ:max</i> is provided in the Verilog description or in the SDF annotation.	
-mindelays	+mindelays
Apply the minimum delay value if a timing triplet in the form <i>min:typ:max</i> is provided in the Verilog description or in the SDF annotation.	
-neg_tchk	+neg_tchk
Allow negative values in \$setuphold and \$recovery timing checks in the Verilog description and in SETUPHOLD and RECREM timing checks in SDF annotation.	
Beginning with the LDV 3.3 release, this is the default. Use -noneg_tchk to disallow the use of negative values.	
-noautosdf	None
Do not perform automatic SDF annotation.	

## NC-Verilog Simulator Help

### SDF Timing Annotation

---

<b>Elaborator (ncelab) options:</b>	<b>Verilog-XL</b>
<code>-no_sdfa_header</code>	None
Do not print messages displaying information contained in the SDF command file.	
<code>-notimingchecks</code>	<code>+notimingchecks</code>
Do not execute timing checks.	
<code>-ntc_warn</code>	<code>+ntc_warn</code>
Print convergence warnings for negative timing checks if delays cannot be calculated given current limit values. By default, warning are not printed.	
<code>-pathpulse</code>	<code>+pathpulse</code>
Enable PATHPULSE\$ specparams, which are used to set module path pulse control on a specific module or on specific paths within modules.	
<code>-pulse_e error_percent</code>	<code>+pulse_e/error_percent</code>
Set the percentage of delay for pulse error limit for both module paths and interconnect. If the <code>-pulse_int_e</code> option is also used, this option applies only to module paths.	
<code>-pulse_int_e error_percent</code>	<code>+pulse_int_e/error_percent</code>
Set the percentage of delay for pulse error limit for interconnect only.	
<code>-pulse_int_r reject_percent</code>	<code>+pulse_int_r/reject_percent</code>
Set the percentage of delay for pulse reject limit for interconnect only.	
<code>-pulse_r reject_percent</code>	<code>+pulse_r/reject_percent</code>
Set the percentage of delay for pulse reject limit for both module paths and interconnect. If the <code>-pulse_int_r</code> option is also used, this option applies only to module paths.	

## NC-Verilog Simulator Help

### SDF Timing Annotation

---

<b>Elaborator (ncelab) options:</b>	<b>Verilog-XL</b>
<code>-sdf_cmd_file</code>	None
Use the specified SDF command file to control SDF annotation.	
<code>-sdf_nocheck_celltype</code>	<code>+sdf_nocheck_celltype</code>
Disable celltype validation between the SDF annotator and the Verilog description. By default, the annotator checks the type specified in the CELLTYPE construct against the module name in the description. If there is a mismatch, a warning is generated and no annotation to that module instance is performed.	
<code>-sdf_no_warnings</code>	<code>+sdf_no_warnings</code>
Do not report warning messages from the SDF annotator.	
<code>-sdf_precision precision</code>	None
Round the precision of timing values in the compiled SDF file.	
<code>-sdf_verbose</code>	<code>+sdf_verbose</code>
Include detailed information in the SDF log file.	
<code>-typdelays</code>	<code>+typdelays</code>
Apply the typical delay value if a timing triplet in the form <code>min:typ:max</code> is provided in the Verilog description or in the SDF annotation.	

---

<b>Simulator (ncsim) Options:</b>	<b>Verilog-XL</b>
<code>-epulse_no_msg</code>	<code>+no_pulse_msg</code>
Suppress pulse control error messages.	

## SDF Annotation for Mixed-Language Designs

In a mixed-language simulation, you can annotate to both the Verilog and VITAL parts of the design from a single SDF file. You can:

- Annotate to both Verilog and VITAL using the SDF command file methodology described in [“VITAL SDF Annotation”](#) on page 790.

For the VITAL part of the design, you must use an SDF command file. You can specify multiple SDF files in the command file, and each of these can go across language boundaries.

In the following example SDF command file, one SDF file is used to annotate to the VHDL scope :pm7324\_inst, which contains Verilog blocks.

```
COMPILED_SDF_FILE = "pm7324_flat.sdf.x",
SCOPE = :pm7324_inst,
LOG_FILE = "pm7324_flat.sdf.log",
MTM_CONTROL = "TYPICAL",
SCALE_FACTORS = "1.0:1.0:1.0",
SCALE_TYPE = "FROM_MTM";
```

- Annotate to the VITAL parts of the design using an SDF command file and annotate to the Verilog parts of the design using \$sdf\_annotation system tasks.

In the current version of the simulator, interconnect delays, including multi-source interconnect delays, are supported across the language boundary except if the language boundary is a bidirect.

## Utilities

---

There are several utilities that are provided with the NC-Verilog simulator:

- *ncexport*

Exports an entire VHDL design hierarchy or just sections of it into a directory structure. See “[ncexport](#)” on page 833 for details on *ncexport*.

- *nchelp*

Provides help on compiler, elaborator, and simulator messages. Used with the `-hdlvar` or `-cdslib` option, *nchelp* prints the contents of the default or of a specified `hdl.var` or `cds.lib` file. See “[nchelp](#)” on page 842 for details on *nchelp*.

- *ncls*

Lists compiled objects stored in the library system and displays various attributes and information about those objects. See “[ncls](#)” on page 847 for details on *ncls*.

- *ncpack*

Lets you change the properties of a packed library database. For example, you can make the database read-only or add-only. See “[ncpack](#)” on page 861 for details on *ncpack*.

- *ncprep*

Provides a quick transition to the NC-Verilog simulator for designs that run in Verilog-XL. This utility is run with the same command-line options you use to run Verilog-XL. It creates everything you need to run the NC-Verilog simulator, including all libraries, a `cds.lib` file, and an `hdl.var` file. The utility also creates a script that you can execute to run the simulator. See “[ncprep](#)” on page 867 for details on *ncprep*.

- *ncrm*

Deletes design units from a library. See “[ncrm](#)” on page 901 for details on *ncrm*.

- *ncsdfc*

Compiles and decompiles SDF files. See “[ncsdfc](#)” on page 906 for details on *ncsdfc*.

■ *ncshell*

Generates a shell file to facilitate model import. See “[ncshell](#)” on page 912 for details on *ncshell*.

■ *ncsuffix*

Displays the machine architecture and the revision number of the library system. See “[ncsuffix](#)” on page 925 for details on *ncsuffix*.

■ *ncupdate*

Automatically recompiles and re-elaborates all out-of-date design units in the hierarchy. See “[ncupdate](#)” on page 928 for details on *ncupdate*.

■ *shellgen*

Generates a Verilog or VHDL model shell that you can then instantiate in a design to import OMI-compliant models. See “[shellgen](#)” on page 934 for details on *shellgen*.

## **ncexport**

The *ncexport* utility copies the source code for an entire compiled or elaborated Verilog, VHDL, or mixed-language design hierarchy into a directory and generates a compilation script. *ncexport* does not automatically execute the script.

For each library in the design to be exported, *ncexport* creates a subdirectory called *library\_name.ncxp* (for example, *worklib.ncxp*). It then copies each design unit in the design into the corresponding directory and gives the unit a file name. The *ncexport* utility uses the following file naming convention:

*<primary\_name>\_<secondary\_name>\_<type>.file\_extension*

where *type* is:

- *m* for Verilog module
- *e* for VHDL entity
- *a* for VHDL architecture
- *p* for VHDL package
- *b* for VHDL package body
- *c* for VHDL configuration

For example, suppose that there are two libraries called *worklib* and *design\_lib* in the design to be exported. *ncexport* creates two subdirectories called *worklib.ncxp* and *design\_lib.ncxp*, and then copies each design unit in the design into the corresponding directory and gives the unit a file name. For example, the *worklib.ncxp* directory might contain files such as the following:

```
test_e.vhd (VHDL entity)
test_vhdl_a.vhd (VHDL architecture)
inc_m.v (Verilog module)
add1_e.vhd (VHDL entity)
add1_vhdl_a.vhd (VHDL architecture)
```

The compilation script that *ncexport* generates depends on the *-target* command-line option. The argument to this option can be *generic*, *inca*, or *synopsys*. See the description of the *-target* option for details.

## **ncexport Command Syntax**

Invoke *ncexport* with options and arguments. Options can occur in any order. Parameters to options must immediately follow the option they modify. Command-line options can be abbreviated to the shortest unique string, indicated here with capital letters.

*ncexport [options] design\_unit*

or:

*ncexport [options] -snapshot snapshot\_name*

where the *design\_unit* or *snapshot\_name* argument is specified in lib.cell:view format. For example,

```
% ncexport -messages -exclude SYNOPSYS -snapshot WORKLIB.TEST:VHDL
```

[-Append log]  
[-CDslib filename]  
[-COntext path\_name]  
[-Directory directory\_name]  
[-ERrormax integer]  
[-EXclude library\_name]  
[-HDLvar filename]  
[-HElp]  
[-Include library\_name]  
[-Logfile logfile\_name]  
[-Messages]  
[-NCError warning\_code]  
[-NCFatal {warning\_code | error\_code}]  
[-NOCopyright]  
[-NOLog]  
[-NOSTdout]  
[-Overwrite]  
[-Snapshot snapshot\_name]  
[-Target target\_name]  
[-Version]

## **ncexport Command Options**

The following list describes the *ncexport* command-line options.

### **-Append\_log**

Append log information from multiple *ncexport* runs into one log file. If you use both `-append_log` and `-nolog` on the command line, `-nolog` overrides `-append_log`.

### **-CDslib *filename***

Use the specified `cds.lib` file. See “[The cds.lib File](#)” on page 110 for information on the `cds.lib` file.

### **-COntext *path\_name***

Use the specified path in the compilation script. The default is the exported directory.

### **-Directory *directory\_name***

Specifies the directory into which *ncexport* copies the VHDL source and writes the compilation script. The default is the current directory. For example, the following command writes the compilation script to the directory called `ncexp`. The source files are copied to `./ncexp/library_name.ncxp`.

```
% ncexport -directory ncexp -snapshot worklib.top:arch
```

### **-ERrormax *integer***

Abort after reaching the specified number of errors. By default, there is no limit on the number of error messages.

### **-Exclude *library\_name***

Exclude the specified library from export. The *ncexport* command excludes STD and IEEE libraries by default.

**-HDIvar *filename***

Use the specified `hdl.var` file. See “[The hdl.var File](#)” on page 118 for information on the `hdl.var` file.

**-HElp**

Display a list of the `ncexport` command-line options with a brief description of each option.

**-Include *library\_name***

Include the specified library for export.

**-LogFile *logfile\_name***

Use the specified name for the log file instead of the default name `ncexport.log`.

You can not include this option in an `hdl.var` file. This option overrides the `-nolog` option.

**-Messages**

Display informational messages during the creation of the compilation script.

**Note:** Information is only written to the log file when you use the `-messages` option.

**-NCError *warning\_code***

Increase the severity level of the specified warning message from warning to error. The `warning_code` argument is the message code (mnemonic) that appears in the warning message following the severity code.

Example:

```
% ncexport -ncerror ABCDEF -directory my_dir my_lib.top:arch
```

You can include multiple `-ncerror` options on the command line.

Using this option can change the behavior of the tool because functions that return errors instead of warnings may behave differently. Warnings that are changed to errors are counted in the error count limit that you specify with the `_errormax` option.

**-NCFatal {*warning\_code* | *error\_code*}**

Increase the severity level of the specified warning message or error message from warning or error to fatal. The *warning\_code* or *error\_code* argument is the message code (mnemonic) that appears in the message following the severity code.

Example:

```
% ncexport -ncfatal LMNOPQ -directory my_dir my_lib.top:arch
```

You can include multiple `-ncfatal` options on the command line.

**-NOCopyright**

Suppress the display of the copyright banner.

**-NOLog**

Do not generate a log file. By default, *ncexport* generates a log file called `ncexport.log`.

**-NOStdout**

Suppress printing of output to the screen.

**-Overwrite**

Overwrite an existing directory with the same name.

**-Snapshot *snapshot\_name***

Specifies the simulation snapshot to export.

**-Target *target\_name***

Specifies the target for the compilation script. The *target\_name* argument can be generic, inca, or synopsys. The default is generic.

If the target is generic, *ncexport* generates a file called `Generic.Dependencies`. This file contains the list of design hierarchy dependencies exported bottom up.

The `-target inca` option generates a script called `Inca.Script` for the Cadence NC simulators.

The `-target synopsys` option generates a script called `Synopsys.Script` in Synopsys format. This argument can only be specified with a pure VHDL design.

### **-Version**

Display the version of `ncexport` and exit.

## **Example ncexport Command Lines**

To export an analysed unit `my_lib.top:arch` into the directory `my_dir`:

```
% ncexport -directory my_dir my_lib.top:arch
```

To export an elaborated unit `my_lib.top:arch` into the directory `my_dir`:

```
% ncexport -directory my_dir -snapshot my_lib.top:arch
```

## **Example**

The example used in this section is a mixed-language design. The top level is VHDL. The file `test.vhd` contains an entity called `test` and an architecture called `vhdl1`. The architecture declares a component called `inc`, which is a Verilog module. This component is then instantiated.

```
-- File test.vhd
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use work.std_conversions.all;

entity test is
end test;

architecture vhdl1 of test is

component inc port (.....);
end component;
...
...
```

## NC-Verilog Simulator Help

### Utilities

---

```
i1 : inc port map (datao, clk, rst );  
...  
...  
end vhdl;
```

The Verilog module `inc` instantiates `add1` and `reg1`, both of which are VHDL entities.

```
// File inc.v  
module inc (.....);  
...  
...  
add1 a3 (.....);  
add1 a2 (.....);  
add1 a1 (.....);  
add1 a0 (.....);  
  
reg1 r3 (.....);  
reg1 r2 (.....);  
reg1 r1 (.....);  
reg1 r0 (.....);  
...  
endmodule
```

The files `add1.vhd` and `reg1.vhd` contain entities called `add1` and `reg1`, respectively. Each entity has an associated architecture called `vhdl`.

```
-- File add1.vhd  
library ieee;  
use ieee.std_logic_1164.all;  
  
entity add1 is  
    port (....);  
end add1;  
  
architecture vhdl of add1 is  
begin  
...  
...  
end vhdl;
```

```
-- File reg1.vhd
library ieee;
use ieee.std_logic_1164.all;

entity reg1 is
    port ( ... );
end reg1;

architecture vhdl of reg1 is
begin
    ...
    ...
end vhdl;
```

The design also includes a VHDL package called `std_conversions` in the file called `std_conversions.vhd`.

```
-- File std_conversions.vhd
library ieee;
use ieee.std_logic_1164.all;

package std_conversions is
    ...
    ...
end std_conversions;

package body std_conversions is
    function to_string (arg : boolean) return string is
        ...
        ...
end std_conversions;
```

To export an elaborated design:

1. Compile and elaborate the design. For example,

```
% ncvlog -messages -work worklib inc.v
% ncvhdl -messages -nobuiltin -work worklib std_conversions.vhd
% ncvhdl -messages -nobuiltin -work worklib test.vhd
% ncvhdl -messages -nobuiltin -work worklib add1.vhd
% ncvhdl -messages -nobuiltin -work worklib reg1.vhd
% ncelab -SDF_VERBOSE WORKLIB.TEST:VHDL
```

In this example, `ncelab` generates a snapshot called `worklib.test:vhd1`.

## NC-Verilog Simulator Help

### Utilities

---

#### 2. Run *ncexport*.

```
% ncexport -messages -directory ncexp -overwrite -target INCA
           -exclude SYNOPSYS -snapshot WORKLIB.TEST:VHDL
ncexport: v03.30.(p001): (c) Copyright 1995 - 2001 Cadence Design Systems, Inc.
ncexport: Started on Jul 30, 2001 at 10:32:54
Export directory 'ncexp'
      created for 'WORKLIB.TEST:VHDL'
      for target 'inca'
ncexport: Exiting on Jul 30, 2001 at 10:32:54 (total: 00:00:00)
```

This command creates a subdirectory called ncexp, which contains the compilation script (Inca.Script) and a directory called worklib.ncxp. This directory contains the following files:

inc_m.v	(Verilog module inc)
std_conversions_p.vhd	(VHDL package std_conversions)
std_conversions_body_b.vhd	(VHDL package body std_conversions)
test_e.vhd	(VHDL entity test)
test_vhdl_a.vhd	(VHDL architecture vhdl of test)
add1_e.vhd	(VHDL entity add1)
add1_vhdl_a.vhd	(VHDL architecture vhdl of add1)
reg1_e.vhd	(VHDL entity reg1)
reg1_vhdl_a.vhd	(VHDL architecture vhdl of reg1)

The Inca.Script file generated by *ncexport* is as follows:

```
#!/bin/csh -f
# ncexport: v03.30.(p001)
# Date : Jul 30, 2001 at 10:32:54

mkdir ./worklib
ncvlog -MESSAGES -WORK worklib ../ncexp/worklib.ncxp/inc_m.v
ncvhdl -MESSAGES -Y -NOBUILTIN -WORK worklib
        ../ncexp/worklib.ncxp/std_conversions_p.vhd
ncvhdl -MESSAGES -Y -NOBUILTIN -WORK worklib ../ncexp/worklib.ncxp/test_e.vhd
ncvhdl -MESSAGES -Y -NOBUILTIN -WORK worklib ../ncexp/worklib.ncxp/test_vhdl_a.vhd
ncvhdl -MESSAGES -Y -NOBUILTIN -WORK worklib
        ../ncexp/worklib.ncxp/std_conversions_body_b.vhd
ncvhdl -MESSAGES -Y -NOBUILTIN -WORK worklib ../ncexp/worklib.ncxp/add1_e.vhd
ncvhdl -MESSAGES -Y -NOBUILTIN -WORK worklib ../ncexp/worklib.ncxp/add1_vhdl_a.vhd
ncvhdl -MESSAGES -Y -NOBUILTIN -WORK worklib ../ncexp/worklib.ncxp/reg1_e.vhd
ncvhdl -MESSAGES -Y -NOBUILTIN -WORK worklib ../ncexp/worklib.ncxp/reg1_vhdl_a.vhd
ncelab -SDF_VERBOSE -MESSAGES -ACCESS +RWC WORKLIB.TEST:VHDL
```

## **nchelp**

Use the *nchelp* utility to:

- Get help on messages generated by the compiler, elaborator, and simulator, and by other utilities.

Syntax:

```
% nchelp [options] tool_name message_code
```

Examples:

```
% nchelp ncvhdl BADCLP  
% nchelp ncvhdl badclp
```

The *nchelp* utility displays extended help on the brief messages generated by the NC-Verilog simulator tools and utilities.

- To print:

- the variables defined in a specific `hdl.var` file or the libraries defined in a specific `cds.lib` file.
  - the variables defined in the default `hdl.var` file or the libraries defined in the default `cds.lib` file.

Syntax:

```
% nchelp [options] {-hdlvar | -cdslib} [filename]
```

Examples:

```
% nchelp -hdlvar  
% nchelp -cdslib ./cds.lib
```

## **nchelp Command Syntax**

Invoke *nchelp* with options and arguments. Options can occur in any order. Parameters to options must immediately follow the option they modify. Command line options can be abbreviated to the shortest unique string, indicated here with capital letters.

```
nchelp [-options] tool_name message_code
  [-All message_code]
  [-Cdslib [filename]]
  [-HDLvar [filename]]
  [-HElp]
  [-NCError warning_code]
  [-NCFatal {warning_code | error_code}]
  [-NEverwarn]
  [-NOCopyright]
  [-NOWarn warning_code]
  [-Tools]
  [-Version]
```

## **nchelp Command Options**

The following list describes the options you can use with the *nchelp* command. Options can be entered in upper or lower case. In the table, capital letters indicate the shortest possible abbreviation for an option.

### **-All *message\_code***

Displays extended help for the message with the specified message code for all tools that can generate the message. For example, the following command displays the help for all tools that can generate the BADCLP error message.

```
% nchelp -all BADCLP
```

### **-Cdslib [*filename*]**

Prints the libraries defined in the specified *cds.lib* file. If no *filename* is specified, prints the libraries defined in the default *cds.lib* file. See “[The cds.lib File](#)” on page 110.

### **-HDLvar [*filename*]**

Prints the variables defined in the specified *hdl.var* file. If no *filename* is specified, prints the variables defined in the default *hdl.var* file. See “[The hdl.var File](#)” on page 118

## **-HElp**

Prints a brief summary of the `nchelp` command options.

## **-NCError *warning\_code***

Increase the severity level of the specified warning message from warning to error. The *warning\_code* argument is the message code (mnemonic) that appears in the warning message following the severity code.

Example:

```
% nchelp -hdlvar -ncerror ABCDEF
```

You can include multiple `-ncerror` options on the command line.

Using this option can change the behavior of the tool because functions that return errors instead of warnings may behave differently.

## **-NCFatal {*warning\_code* | *error\_code*}**

Increase the severity level of the specified warning message or error message from warning or error to fatal. The *warning\_code* or *error\_code* argument is the message code (mnemonic) that appears in the message following the severity code.

Example:

```
% nchelp -hdlvar -ncfatal LMNOPQ
```

You can include multiple `-ncfatal` options on the command line.

## **-NEverwarn**

Disables printing of all warning messages.

## **-NOCopyright**

Suppresses printing of the copyright banner.

**-NOWarn *warning\_code***

Disables the printing of the specified warning. The *warning\_code* is the sequence of characters following the error severity code.

**-Tools**

Displays a list of all of the tools for which extended help is available.

**-Version**

Prints the version of *nchelp* and exits.

**Example ncelp Command Lines**

The following command provides help on the BADCLP message from *ncvlog*.

```
% ncelp ncvlog BADCLP
ncelp: v1.0.(p2): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
ncvlog/BADCLP = The path specified for the -CDSLIB argument is invalid, please check
that the specified path exists and is readable.
%
```

The following command provides help on the CUVWSP message from *ncelab*.

```
% ncelp ncelab cuvwsp
ncelp: v1.0.(p2): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
ncelab/CUVWSP = The indicated module instance contains a connection list that does
not connect all the module ports.
%
```

The following command provides help on the NOSNAP message from *ncsim*.

```
% ncelp ncsim NOSNAP
```

## NC-Verilog Simulator Help

### Utilities

---

The following command prints the variables defined in the default hdl.var file. In this example, the default hdl.var file is in the user's home directory (~larry/hdl.var) and the file contains a SOFTINCLUDE statement to include a second hdl.var file.

```
% nchelp -hdlvar
nchelp: v1.0.(p2): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
Parsing -HDLVAR file ./hdl.var.
hdl.var files:
```

```
1: ./hdl.var

2: ./inca/adder/hdl.var

included on line 4 of ./hdl.var
```

Variables defined:

Defined in ./hdl.var:

Line #	Name	Value
-----	-----	-----
1	NCVLOGOPTS	-messages
2	NCELABOPTS	-messages
3	NCSIMOPTS	-messages -tcl

Defined in ./inca/adder/hdl.var:

Line #	Name	Value
-----	-----	-----
1	WORK	worklib

The following command prints the variables defined in ~larry/inca/adder/hdl.var.

```
% ncelp -hdlvar ~larrybird/inca/adder/hdl.var
ncelp: v1.0.(p2): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
Parsing -HDLVAR file ./inca/adder/hdl.var.
```

hdl.var files:

```
1: ./inca/adder/hdl.var
```

Variables defined:

Defined in ./inca/adder/hdl.var:

Line #	Name	Value
-----	-----	-----
1	WORK	worklib

## ncls

The *ncls* utility lists the compiled objects that are stored in the library system and displays various attributes and information about those objects.

When you compile and elaborate a design, all intermediate objects that are required by the INCA tools are logically and physically stored in a library system. The interface to this library system is through the `cds.lib` file, which defines the libraries by associating logical library names with physical library directories. Intermediate objects are stored in a single database file in a library directory. This library database file is called `inca.architecture.lib_version.pak`. For example, the name of the library database file is similar to the following:

```
inca.sun4v.091.pak  
inca.hppa.091.pak
```

In virtually all cases, you can treat the contents of a library system as a black box. If, however, you need to list the objects in the library system, the *ncls* utility provides this visibility.

**Note:** *ncls* only lists those objects that are generated and used by the INCA core system. The utility does not list any other objects that may be stored in the same libraries with the INCA core data.

### **Listing Objects**

Each object in the library system has a three-part name called a *lib.cell:view*. In addition to this three-part name, INCA core data can be categorized into two logical subgroups by object type and by design unit type.

- The elements of the object type group are Verilog Syntax Tree (VST), VHDL Syntax Tree (AST), Overlay Table (SIG), Code (COD), and Simulation Snapshot (SSS).
- The elements of the design unit type group are Verilog module, Verilog primitive, VHDL entity, VHDL architecture, VHDL package, VHDL package body, VHDL configuration, and Simulation Snapshot.

Any object in the library system can be uniquely described by its *lib.cell:view* name plus its object type and design unit type. The *ncls* utility lists the INCA core data objects in the following format:

```
design_unit_type lib.cell[:view] (object_type)
```

**Examples:**

```
module worklib.top:verilog (VST)
entity asic1.flop (AST)
snapshot worklib.main:snap (SSS)
```

*nc/s* sorts the data before displaying it. For example, the utility groups together all of the data from the same library, all of the data from the same cell, and all of the data from the same view. All object types that are built from other object types are displayed in build order.

The *nc/s* command has many options that let you specify the objects that you want to list. You can select objects by name, by object type, by design unit type, or by a combination of these three. The following sections summarize the *nc/s* command-line options. See “[“nc/s Command Options”](#) on page 853 for details on these command-line options.

### Selecting Objects by Name

You can select objects to display by specifying one or more [*lib.*][*cell*][:*view*] arguments. Each of the bracketed items is optional.

This naming convention assumes that a name without any punctuation (that is, without a period or a semi-colon) is a cell name. For example, in the following command, *nc/s* interprets *drink\_machine* as a cell name.

```
% nc/s drink_machine
```

If you want to display all of the elements in a library or all of the elements with a particular view name, include the punctuation or use the *-library* or *-view* option.

- *-library*—Modifies the meaning of an argument without any punctuation from all of the objects that match this cell name to all objects that match this library name.
- *-view*—Modifies the meaning of an argument without any punctuation from all of the objects that match this cell name to all objects that match this view name.

For example, the following commands are identical. They both specify that you want to display all objects that match the view name *rtl*.

```
% nc/s :rtl
% nc/s rtl -view
```

Use the *-all* option to specify a listing of all objects. For example:

```
% nc/s -all
```

The [*lib.*][*cell*][:*view*] names can be in any of the accepted name spaces that are allowed by the INCA tools (Verilog, NVerilog, VHDL, Filesys, and Library). The *nc/s* utility lists any objects that it finds in the library system that match the command-line argument in any

## NC-Verilog Simulator Help

### Utilities

---

of these name spaces. See the NMP documentation for a description of the exact meaning of the name spaces.

#### **Selecting Objects by Object Type**

To display only information about a particular object type, use one or more of the following options:

---

<b>Option</b>	<b>Object Type</b>
-verilog	Verilog Syntax Tree (VST)
-vhdl	VHDL Syntax Tree (AST)
-overlay	Overlay Table (SIG)
-code	Code (COD)
-snapshot	Simulation Snapshot (SSS)

---

#### **Selecting Objects by Design Unit Type**

To display only information about a particular design unit type, use one or more of the following options:

---

<b>Option</b>	<b>Design Unit Type</b>
-module	Verilog modules
-primitive	Verilog primitives
-entity	VHDL entities
-architecture	VHDL architectures
-package	VHDL packages
-body	VHDL package bodies
-configuration	VHDL configurations
-snapshot	simulation snapshots

---

## Display Options

There are several options that you can use on the `ncls` command to display additional information about objects.

Option	Function
<code>-messages</code>	Controls the printing of attributes about the objects.
<code>-command</code>	Prints the command line used to generate the specified object (applies only to AST/VST/SSS objects).
<code>-source</code>	Prints the source files that were used in the compilation of the specified object (only applies to AST/VST/SSS objects).
<code>-dependents</code>	Prints the dependencies that the object has to other INCA core data objects.
<code>-time</code>	Prints the timestamp that is associated with an object.

## **ncls Command Syntax**

Invoke *nc/s* with options and arguments. Options can occur in any order. Parameters to options must immediately follow the option that they modify. Command-line options can be abbreviated to the shortest unique string, indicated here with capital letters.

You can set the NCLSOPTS environment variable or use the NCLSOPTS variable in the hdl.var file to include ncls command-line options.

```
ncls [options] [lib.][cell][:view]
      [-ALL]
      [-APPend_log]
      [-ARchitecture]
      [-Body]
      [-CDslib filename]
      [-CODE]
      [-COMmand]
      [-CONfiguration]
      [-Dependents]
      [-Entity]
      [-File filename]
      [-HDLvar filename]
      [-HELP]
      [-Library]
      [-LOGfile logfile_name]
      [-MESSages]
      [-MOdule]
      [-NCError warning_code]
      [-NCFatal {warning_code | error_code}]
      [-NEverwarn]
      [-NOCopyright]
      [-NOLog]
      [-NOSTdout]
      [-NOWarn warning_code]
      [-Overlay]
      [-PAckage]
      [-PRimitive]
      [-SNapshot]
      [-SOurce]
      [-Time]
```

## **NC-Verilog Simulator Help**

### Utilities

---

[ -VERIlog ]

[ -VERSion ]

[ -VHdl ]

[ -VIew ]

## **ncls Command Options**

The following list describes the options that you can use with the `ncls` command. You can enter options in upper or lower case. In the table, capital letters indicate the shortest possible abbreviation for an option.

### **-ALI**

List all objects in all libraries.

If you use `-all` and a `[lib.][cell][:view]` argument on the command line, `ncls` ignores the `-all` option.

### **-APPend\_log**

Append log information from multiple runs of `ncls` to one log file. This option is overridden by the `-nolog` option.

### **-ARchitecture**

List VHDL architecture objects.

### **-Body**

List VHDL package body objects.

### **-CDslib *filename***

Use the specified `cds.lib` file. See “[The cds.lib File](#)” on page 110 for information on the `cds.lib` file.

### **-CODO**

List Code (COD) objects.

### **-COMmand**

Print the command line used to generate the specified object. This option applies only to AST, VST, and SSS objects.

**Note:** This option does not apply to objects generated by *ncverilog*.

The command line that *nc/s* displays includes all arguments that are required to compile the object. This command line may be different from the original command line used to compile the object. Specifically, you may need some additional options in order to force compilation to the matching library and view and to find the correct *cds.lib* and *hdl.var* files. Because of this, you should use the command line for informative purposes only. You should not use it to regenerate the object. See “[ncupdate](#)” on page 928 (especially the documentation for the *-script* and *-force* options) to find correct command lines for regeneration of an object.

### **-COnfiguration**

List VHDL configuration objects.

### **-Dependents**

Show the dependencies that the specified object has to other INCA core data objects. For example, the following command displays all of the Verilog and VHDL objects that were used to build a snapshot.

```
% nc/s -snapshot -dependents snap_name
```

### **-Entity**

List VHDL entity objects.

### **-File *filename***

Use the command-line arguments contained in the specified file.

You can store frequently used or lengthy command lines by putting command options and arguments in a text file. When you invoke *nc/s* with the *-file* option, the arguments in the specified file are used with the command as if you had entered them on the command line.

### **-HDIvar *filename***

Use the specified *hdl.var* file. See “[The hdl.var File](#)” on page 118 for information on the *hdl.var* file.

### **-HElp**

Display a brief summary of the `ncls` command-line options.

### **-LLibrary**

Indicates that the name specified on the command line is a library.

If you specify a name as an argument on the command line with no period or semi-colon (for example, % `ncls worklib`), *nc/s* interprets the name as a cell name. The `-library` option specifies that the name is a library name.

### **-LOgfile *logfile\_name***

Use the specified name for the log file instead of the default name `ncls.log`.

Use `-nolog` if you don't want a log file. If you use both `-logfile` and `-nolog` on the command line, `-logfile` overrides `-nolog`.

Use `-append_log` if you are going to run *nc/s* multiple times and you want all of the log information appended to one log file. If you do not use this option, *nc/s* overwrites the log file each time you run *nc/s*.

### **-MEssages**

Display attributes about the objects in addition to listing them. The attributes that are currently printed are as follows:

- [ ### bytes ]

Indicates the size of the object.

- [ TMP ]

Indicates that the object is physically in the TMP area for a library. See “[The cds.lib File](#)” on page 110 for information on assigning the TMP area attribute.

- [ unreadable ]

Indicates that the object is unreadable by the INCA tools. This is usually due to dependency recompilation or deletion. For example, this attribute is listed for a snapshot if one or more of the VST files have been recompiled or deleted.

■ [out-of-date]

Indicates that the object is not up-to-date. This is probably caused by one or more source files or INCA core data objects being newer or older than expected.

■ [saved]

Indicates that the snapshot being displayed was saved from the simulator (that is, the snapshot is not a time zero snapshot generated by the elaborator).

■ [MRA]

Indicates that this is the most recently analyzed architecture for a given VHDL entity. This architecture will be the one selected for default VHDL binding.

## **-Module**

List Verilog module objects.

### **-NCError *warning\_code***

Increase the severity level of the specified warning message from warning to error. The *warning\_code* argument is the message code (mnemonic) that appears in the warning message following the severity code.

Example:

```
% ncls -all -ncerror ABCDEF
```

You can include multiple `-ncerror` options on the command line.

Using this option can change the behavior of the tool because functions that return errors instead of warnings may behave differently.

### **-NCFatal {*warning\_code* | *error\_code*}**

Increase the severity level of the specified warning message or error message from warning or error to fatal. The *warning\_code* or *error\_code* argument is the message code (mnemonic) that appears in the message following the severity code.

Example:

```
% ncls -all -ncfatal LMNOPQ
```

You can include multiple `-ncfatal` options on the command line.

**-NEverwarn**

Disable the printing of all warning messages.

To turn off one or more specific warning messages, use [-nowarn](#).

**-NOCopyright**

Suppress printing of the copyright banner.

**-NOLog**

Do not generate a log file. By default, *nc/s* generates a log file called `nc/s.log`.

If you use both `-nolog` and `-logfile` on the command line, `-logfile` overrides `-nolog`.

**-NOStdout**

Suppress printing of output to the screen.

**-NOWarn *warning\_code***

Disable printing of the warning that has the specified code. The *warning\_code* argument is the message code (mnemonic) that appears in the warning message following the error severity code.

You can include multiple `-nowarn` options on the command line.

**-Overlay**

List Overlay Table (SIG) objects.

**-PAckage**

List VHDL package objects.

### **-PRimitive**

List Verilog primitive objects.

### **-SNAPSHOT**

List snapshot (SSS) objects.

### **-SOURCE**

Print the source files that were used in the compilation of the object. This option only applies to AST, VST, and SSS objects.

The source files that are listed for a snapshot include all of the source files that were used by all dependents (ASTs and VSTs). You can use this option to get a list of all the necessary files for a test case (with the exception of any files used during simulation, such as \$readmem).

For Verilog and VHDL objects, the source file information includes the line numbers used in the individual source files. This can be useful for Verilog in finding cross-file inheritance dependencies. Note that if the object is out-of-date, the line numbers reflect the original positions in the source file, not the current positions.

### **-TIME**

Print the timestamp associated with an object. This timestamp is the time at which the object was created.

If you use this option with the `-source` or `-dependents` option, the times that are displayed for the source files or dependents are the expected timestamps for those files. This can help you to understand why a particular object is out-of-date. However, to determine exactly why an object is considered out-of-date, run `ncupdate` with the `-norecompile` and `-verbose` options.

### **-VERILOG**

List Verilog (VST) objects.

## **NC-Verilog Simulator Help**

### Utilities

---

#### **-VERSiOn**

Print the version of *ncls* and exit.

#### **-VHdI**

List VHDL (AST) objects.

#### **-View**

Indicates that the name specified on the command line is a view name.

If you specify a name as an argument on the command line with no period or semi-colon (for example, % *ncls rtl*), *ncls* interprets the name as a cell name. The **-view** option specifies that the name is a view name.

## Example ncls Command Lines

The following command lists all objects in all libraries.

```
% ncls -all
```

The following command lists all Verilog objects named top.

```
% ncls -verilog top
```

The following command lists all VHDL architectures named ARCH\_1.

```
% ncls -vhdl -architecture ARCH_1
```

The following command lists all of the objects in library asic\_1.

```
% ncls -library asic_1
```

The following command displays information on all VHDL entities and all Verilog modules and primitives.

```
% ncls -verilog -entity
```

All of the following commands can be used to list the snapshot worklib.top:snap.

```
% ncls -snapshot worklib.top:snap
% ncls -snapshot worklib.top
% ncls -snapshot top:snap
% ncls -snapshot top
% ncls -view -snapshot snap
% ncls -view snap
```

The following command includes the -source option. This prints the source files that are used by the snapshot worklib.top:snap. The source files listed for the snapshot include all source files used by all dependents (ASTs and VSTs).

```
% ncls -source -snapshot worklib.top:snap
```

The following command lists the source files that were used in compiling worklib.dut:v.

```
% ncls -source -verilog worklib.dut:v
```

The following command lists the objects that were used in building the snapshot top:snap.

```
% ncls -dependents -snapshot worklib.top:snap
```

The following command includes the -command option. This option shows the command line that was used to build the object my\_module.

```
% ncls -command my_module
```

## **ncpack**

The *ncpack* utility lets you change the properties of a packed library database. When you compile source files or elaborate a design, packed library databases are generated. These databases are readable and writable by default. You can:

- Make a library database read-only (*-readonly*).
- Make a library database add-only (*-addonly*).

New data can be added to an add-only database, but existing data cannot be removed or modified.

- Make a library database readable/writable (*-database*).

You can use *ncpack* with the *-unpack* option to unpack a packed library so that you can see which cells and views are in the library. If you then repack the library, *ncpack* creates a read-only database.

## **ncpack Command Syntax**

Invoke *ncpack* with options and arguments. Options can occur in any order. Parameters to options must immediately follow the option they modify. Command-line options can be abbreviated to the shortest unique string, indicated here with capital letters.

```
ncpack [-options] logical_library_name ...  
[-ADdonly]  
[-APPend log]  
[-Cdslib filename]  
[-Database]  
[-Errormax integer]  
[-HDLvar filename]  
[-HELP]  
[-Loqfile filename]  
[-Messages]  
[-NCError warning_code]  
[-NCFatal {warning_code | error_code}]  
[-NEverwarn]  
[-NOCopyright]  
[-NOLog]  
[-NOSTdout]  
[-NOWarn warning_code]  
[- Readonly]
```

[-Tmpdir *directory*]  
[-UNLock]  
[-UNPack]  
[-Version]

## **ncpack Command Options**

The following list describes the options you can use with the *ncpack* command. Options can be entered in upper or lower case. Capital letters indicate the shortest possible abbreviation for an option.

### **-ADdonly**

Makes the packed library database an add-only database. New data can be added to an add-only database, but nothing in the database can be deleted or modified. It is recommended that you mark shared libraries as add-only.

### **-APPend\_log**

Appends log information from multiple runs of *ncpack* to one log file. This option is overridden by the *-nolog* option.

### **-Cdslib *filename***

Uses the specified *cds.lib* file. See “[The cds.lib File](#)” on page 110.

### **-Database**

Generates a writeable packed library.

If you unpack a library and then repack it, *ncpack* creates a read-only library. Use the *-database* option to make the library writeable.

### **-Errormax *integer***

Aborts *ncpack* after reaching the specified number of errors.

**-HDIvar *filename***

Uses the specified `hdl.var` file. See “[The hdl.var File](#)” on page 118.

**-HElp**

Prints a brief summary of the `ncpack` command-line options.

**-LogFile *filename***

Uses the specified name for the log file instead of the default name `ncpack.log`.

**-Messages**

Prints informative messages during execution.

**-NCError *warning\_code***

Increase the severity level of the specified warning message from warning to error. The `warning_code` argument is the message code (mnemonic) that appears in the warning message following the severity code.

Example:

```
% ncpack -messages -database worklib -nerror ABCDEF
```

You can include multiple `-nerror` options on the command line.

Using this option can change the behavior of the tool because functions that return errors instead of warnings may behave differently.

**-NCFatal {*warning\_code* | *error\_code*}**

Increase the severity level of the specified warning message or error message from warning or error to fatal. The `warning_code` or `error_code` argument is the message code (mnemonic) that appears in the message following the severity code.

Example:

```
% ncpack -messages -database worklib -ncfatal LMNOPQ
```

You can include multiple `-ncfatal` options on the command line.

**-NEverwarn**

Disables printing of all warning messages.

**-NOCopyright**

Suppresses printing of the copyright banner.

**-NOLog**

Do not generate a log file. By default, *ncpack* generates a log file called *ncpack.log*.

**-NOStdout**

Suppresses printing output to the screen.

**-NOWarn *warning\_code***

Disables the printing of the specified warning. The *warning\_code* is the sequence of characters following the error severity code.

**- Readonly**

Makes the packed database read-only. The database generated by compiling and elaborating a design is read/write by default. Use the *-readonly* option to make the database read-only.

**-Tmpdir *directory***

Specifies the directory that *ncpack* uses to store temporary data while it packs the library. The temporary directory can be located anywhere in the local file system or on a remote machine.

When a library is being packed or unpacked it temporarily uses about two times the disk space required by the library. If the disk on which you are operating is not large enough to handle this much additional data, you can use the *-tmpdir* option to specify a different directory to store the temporary data.

**Note:** This option is not the same as assigning the **TMP** attribute in *cds.lib*.

**-UNLock**

Unlocks the specified library.

When you compile source files or elaborate a design, a lock is placed on the database as it is being generated. Use the `-unlock` option to remove the lock to make the database writeable immediately.

Locks are placed on databases to prevent different users from trying to write to them simultaneously. The `-unlock` option should only be used in situations where a locked database *must* be unlocked. For example, a suspended process can keep a database locked, and you may want to use `-unlock` to unlock that database.

**-UNPack**

Unpacks the specified library.

**-Version**

Prints the version of *ncpack* and exits.

## Example ncpack Command Lines

The following command makes the packed library database for library `worklib` read-only.

```
% ncpack -readonly worklib
```

The following command makes the packed library database for library `worklib` and for library `alt_max2` add-only.

```
% ncpack -addonly worklib alt_max2
```

The following command unpacks the data stored in the library database file (.pak) in the library named `worklib`. A subdirectory under the `worklib` directory is created for each cell and for each view, and the appropriate intermediate objects are placed in the subdirectories. The packed library is then removed.

```
% ncpack -unpack -messages worklib
```

```
ncpack: v2.2.(d5): (c) Copyright 1995-1999 Cadence Design Systems, Inc.
```

```
    Unpacking library 'worklib' ...
    Moving packed library to temporary location ...
    Reading packed library index ...
    Copying packed objects back into library ...
    Removing packed library ...
```

```
% ls worklib
```

```
board/           dEdgeFF/          m16/           m555/
```

The following command packs the library named `worklib`. The `-database` option is included to make the database writable.

```
% ncpack -messages -database worklib
```

```
ncpack: v2.2.(d5): (c) Copyright 1995-1999 Cadence Design Systems, Inc.
```

```
    Packing library 'worklib' ...
    Scanning library ...
    Creating packed library index ...
    Copying objects into packed library ...
    Writing packed library index ...
    Removing packed objects from library ...
    Moving packed library into place ...
```

```
% ls worklib
```

```
board/           inca.sun4v.090.pak      m555/
dEdgeFF/          m16/
```

## ncprep

*ncprep* is a utility that provides a quick transition to the NC-Verilog simulator for designs that already run in Verilog-XL. This utility is run with the same command-line options you use to run Verilog-XL. The *ncprep* utility evaluates the Verilog-XL command options and generates script and data files that contain the corresponding NC-Verilog simulator options.

*ncprep* creates the following files and directories:

- A library directory called `INCA_libs`
- A work directory called `INCA_libs/worklib`
- A `cds.lib` file that defines all of the libraries. See “[The cds.lib File](#)” on page 110 for details on the `cds.lib` file.
- An `hdl.var` file. See “[The hdl.var File](#)” on page 118 for details on the `hdl.var` file.
- `ncvlog.args`—A file of translated arguments that is passed to the NC-Verilog parser, *ncvlog*.
- `ncelab.args`—A file of translated arguments that is passed to the NC-Verilog elaborator, *ncelab*.
- `ncsim.args`—A file of translated arguments that is passed to the NC-Verilog simulator, *ncsim*.
- A log file (called `ncprep.log` by default).
- `RUN_NC`—A script file that contains commands to run *ncvlog*, *ncelab*, and *ncsim* using the arguments in the corresponding `.args` files.

If the files already exist, *ncprep* reports an error and stops. To overwrite existing files, use the `+overwrite` option to *ncprep*.

After you have run *ncprep*, you can execute the `RUN_NC` script. The script uses the `*.args` files to run *ncvlog*, *ncelab*, and *ncsim* to compile, elaborate, and simulate your design.

The `ncvlog.args` file lists all of the HDL files required for a particular design. The *ncprep* utility searches library directories and only includes the files that are needed from the directories. However, *ncprep* can't determine which modules are required in the library files, so it includes the entire file in the `ncvlog.args` file. Thus *ncvlog* compiles the complete library file even though only a few modules are needed.

*ncprep* cannot convert every Verilog-XL option to an equivalent NC-Verilog simulator option. Some designs will still require manual intervention. Also, *ncprep* is not designed to create

an optimized simulation flow. Your coding style, library management process, and design environment will define the ultimate performance of the simulator.

By default, the elaborator (*ncelab*) creates a simulation snapshot in which simulation objects are tagged as having no read, write, or connectivity access. This increases the performance of the simulator for long regression test runs, but does not provide the access to objects that you need to debug a design. There are two *ncprep* command-line options you can use to specify access to simulation objects:

- [+debug](#). This option turns on read access to all objects in the design. Read access is required for probing nets, regs and variables (including setting PLI callbacks) and getting the value of these objects. The *ncprep* [+debug](#) option is translated to the *ncelab* [-access +r](#) option.
- [+linedebug](#). This option enables support for setting line breakpoints and for single-stepping through code, and provides read, write, and connectivity access to all objects in the design. The [+linedebug](#) option is translated to the *ncvlog* [-linedebug](#) option.

You can also edit the *ncelab.args* file to insert the [-access](#) or [-afile](#) elaborator command options.

## ncprep Command Syntax

Invoke *ncprep* with the *ncprep* command followed by the Verilog-XL command-line arguments and any *ncprep* command options.

```
ncprep [ncprep_options] verilog-xl_arguments
  [+debug]
  [-h]
  [-l filename]
  [+linedebug]
  [+ncerror+warning\_code]
  [+ncfatal+{warning\_code | error\_code}]
  [+nclibdirname+directory\_name]
  [+overwrite]
  [+redirect+directory\_name]
```

## **ncprep Command Options**

The following list describes the `ncprep` command-line options:

### **+debug**

Create a simulation snapshot in which all simulation objects are tagged as having read access. This option is translated to the `ncelab -access +r` option. See “[Enabling Read, Write, or Connectivity Access to Simulation Objects](#)” on page 248 for details on specifying access.

### **-h**

Display help on the `ncprep` command and its options.

### **-l *filename***

Use the specified name for the log file instead of the default name, `ncprep.log`.

`ncprep` ignores the Verilog-XL `-l filename` option and generates a log file which is called `ncprep.log` by default. In the `ncvlog.args`, `ncelab.args`, and `ncsim.args` output files, `ncprep` inserts a `-logfile` option. For example, in the `ncvlog.args` file, `ncprep` inserts the following lines:

```
// Uncomment the following line to generate a named log file  
// -LOGFILE ncvlog.log
```

Uncomment these lines if you want a log file when you run the `RUN_NC` script.

### **+linedebug**

Enable support for setting line breakpoints and for single-stepping through code. Using this option automatically provides read, write, and connectivity access. This option is translated to the `ncvlog -linedebug` option.

### **+ncerror+warning\_code**

Increase the severity level of the specified warning message from warning to error. The `warning_code` argument is the message code (mnemonic) that appears in the warning message following the severity code.

## NC-Verilog Simulator Help

### Utilities

---

Example:

```
% ncprep -f verilog.vc +ncerror+CUVWSP
```

You can include multiple `+ncerror+` options on the command line.

Using this option can change the behavior of the tool because functions that return errors instead of warnings may behave differently. Warnings that are changed to errors are counted in the error count limit that you specify with the `+max_err_count+` option.

#### **`+ncfatal+{warning_code | error_code}`**

Increase the severity level of the specified warning message or error message from warning or error to fatal. The `warning_code` or `error_code` argument is the message code (mnemonic) that appears in the message following the severity code.

Example:

```
% ncprep -f verilog.vc +ncfatal+DLCPTH
```

You can include multiple `-ncfatal` options on the command line.

#### **`+nclibdirname+directory_name`**

Store the snapshot, packed library file, and other generated objects in a directory with the specified name.

By default, *ncprep* creates a directory called `INCA_libs` in the current working directory to store these objects. Use the `+nclibdirname+` option to specify a different directory. *ncprep* automatically creates the directory for you.

The `directory_name` argument can be a relative or absolute path to the directory. For example:

```
+nclibdirname+foo          // Stores objects in ./foo

+nclibdirname./foo         // Stores objects in ./foo

+nclibdirname+foo/bar      // Stores objects in ./foo/bar. Directory foo must
                           // exist. ncprep will create directory bar.

+nclibdirname./foo/bar     // Stores objects in ./foo/bar

+nclibdirname../foo        // Stores objects in ../foo
```

## NC-Verilog Simulator Help

### Utilities

---

```
+nclibdirname+../foo/bar      // Stores objects in ../foo/bar  
  
+nclibdirname+/tmp            // Stores objects in /tmp  
  
+nclibdirname+..               // Stores objects in .. (upper directory)
```

Use the `+redirect+` option if you want all *ncprep* output redirected to a specific directory.

#### **+overwrite**

Overwrite any script files, setup files, and directories that already exist. The default is to print an error message and not overwrite files or directories. You cannot abbreviate `+overwrite`.

#### **+redirect+*directory\_name***

Redirect the output of *ncprep* to the specified directory. This option creates a directory with the specified name, and uses that directory to store the following output: the `INCA_libs` directory, the `ncvlog`, `ncelab`, and `ncsim` arguments files, the `cds.lib` and `hdl.var` files, and the `RUN_NC` script.

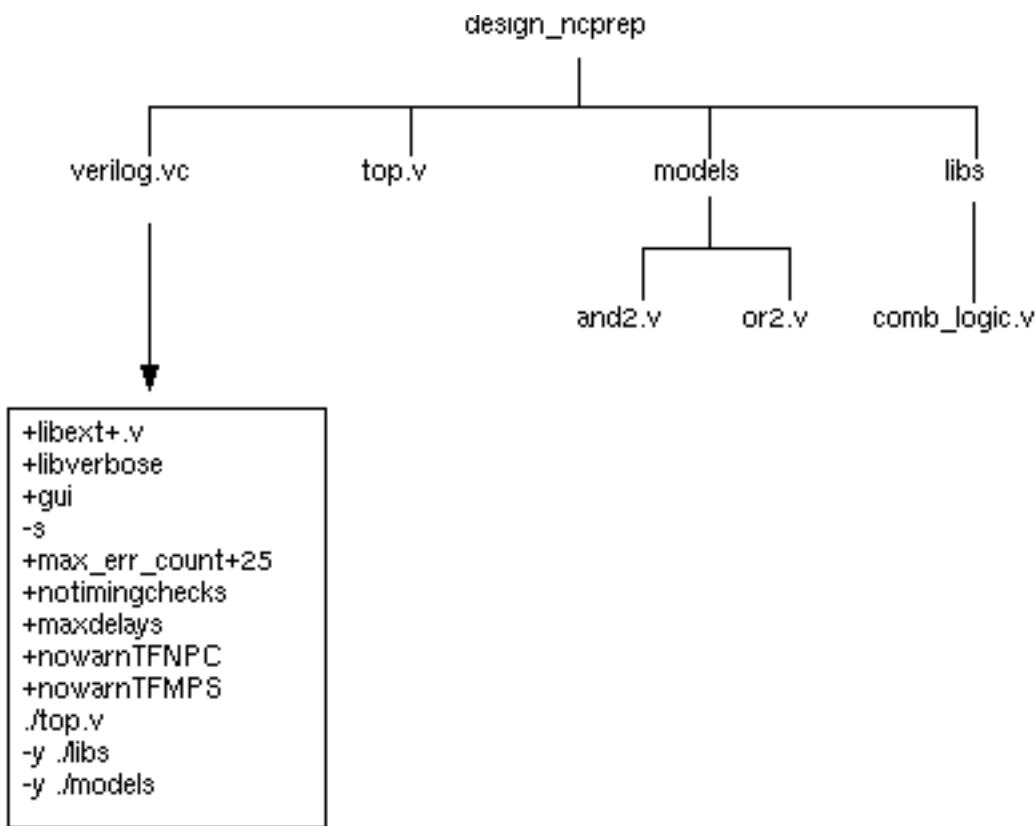
The `ncprep.log` file is stored in the current working directory.

## NC-Verilog Simulator Help

### Utilities

#### Example ncprep Run

In the following example, source files are in the directory structure shown below. Click on [top.v](#) to see the source files used in the example.



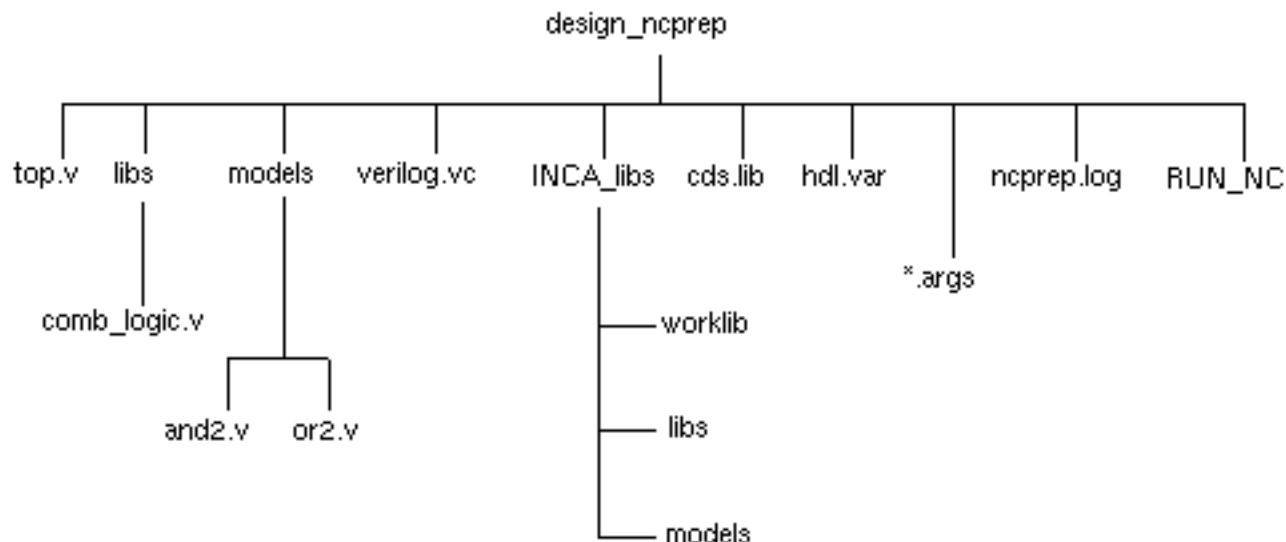
```
% ncprep -f verilog.vc
ncprep: v2.2.(d6): (c) Copyright 1995 - 1999 Cadence Design Systems, Inc.
Translating +libverbose to ncelab option -LIBVERBOSE
Translating +gui to ncsim option -GUI
Translating +max_err_count+25 to ncvlog, ncelab, ncsim option ERRORMAX
Translating +notimingchecks to ncelab option -NOTIMINGCHECKS
Translating +maxdelays to ncelab option -MAXDELAYS
Translating +nowarnTFNPC to ncvlog, ncelab, ncsim option -NOWARN
Translating +nowarnTFMPS to ncvlog, ncelab, ncsim option -NOWARN
Adding -UPDATE option, to disable update run ncprep with the +noupdate
$sdff_annotation() system task detected in design. Pass +noautosdf option to ncprep,
to turn off Automatic SDF annotation.
Translation successful.
```

## NC-Verilog Simulator Help

### Utilities

---

This run of *ncprep* creates the following directory structure:



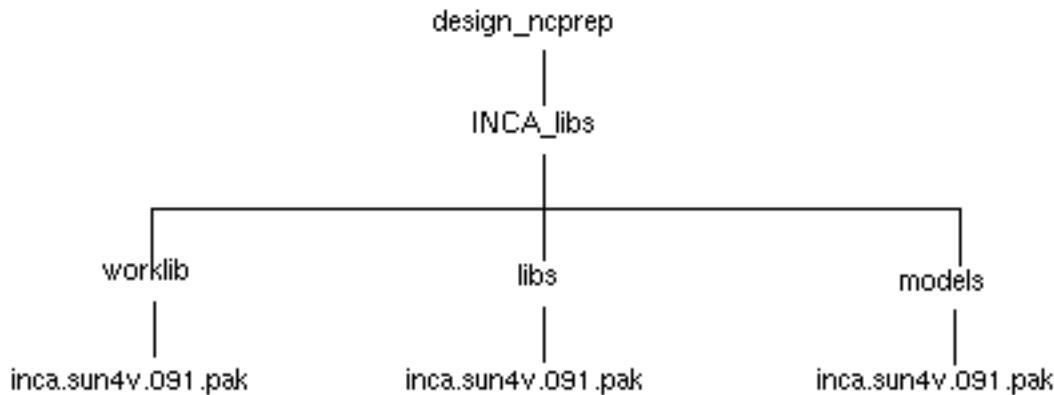
After running *ncprep*, you can execute the *RUN\_NC* script to compile, elaborate, and simulate the design. The following shows the *RUN\_NC* script:

```
#!/bin/csh -f
#
# RUN_NC: Script to run all of NC-Verilog (the first time).
# Created by ncprep on Thu May 13 10:08:29 1999
#
# Run the NC-Verilog parser (compile the source)
ncvlog -f ncvlog.args
if ($status != 0) then
    exit
endif
# Run the NC-Verilog elaborator (build the design hierarchy)
ncelab -f ncelab.args
if ($status != 0) then
    exit
endif
# Run the NC-Verilog simulator (simulate the design)
ncsim -f ncsim.args
```

When you execute the *RUN\_NC* script, log files for the three tools are created (*ncvlog.log*, *ncelab.log*, and *ncsim.log*) in the current working directory.

After you execute the *RUN\_NC* script for this example, the *INCA\_libs* directory contains the following directories and files. The *.pak* files are packed library database files. These

databases are readable and writable by default. Use the *ncpack* utility to change the properties of the databases. For example, you can use *ncpack* to make the databases read-only or add-only. See “[ncpack](#)” on page 861 for details on *ncpack*.



## Example ncprep Output Files

The following sections show the output files created by running the example in “[Example ncprep Run](#)” on page 872.

### ***ncprep* Output File—cds.lib**

*ncprep* creates a `cds.lib` file in the current run directory. See “[The cds.lib File](#)” on page 110 for details on the `cds.lib` file.

```
# cds.lib: Defines the locations of compiled libraries.
# Created by ncprep on Thu May 13 11:27:25 1999
#
DEFINE worklib ./INCA_libs/worklib
DEFINE libs ./INCA_libs/libs          // These libraries are specified in the XL
                                         // arguments file, verilog.vc.
DEFINE models ./INCA_libs/models
```

### ***ncprep Output File—hdl.var***

*ncprep* creates an `hdl.var` file in the current run directory. See “[The hdl.var File](#)” on page 118 for details on the `hdl.var` file.

```
# hdl.var: Defines variables used by the INCA tools.  
# Created by nprep on Thu May 13 11:27:25 1999  
#  
softinclude $CDS_INST_DIR/tools/inca/files/hdl.var  
define LIB_MAP ( $LIB_MAP, + => worklib )  
define VIEW_MAP ( $VIEW_MAP, .v => module_v)  
define LIB_MAP ( $LIB_MAP, ./libs => libs )  
define LIB_MAP ( $LIB_MAP, ./models => models )
```

### ***ncprep Output File—ncvlog.args***

*ncprep* creates the `ncvlog.args` file. This file is the arguments file for the `ncvlog` command. See [Chapter 7, “Compiling Verilog Source Files with ncvlog.”](#) for details on `ncvlog`.

```
// ncvlog.args: Arguments passed to the NC-Verilog parser.  
// Created by nprep on Thu May 13 12:53:53 1999  
-ERRORMAX 25  
  
// Frequently occurring Verilog Warnings.  
  
// Turn on informative messages.  
-MESSAGES  
// -NOCOPYRIGHT  
  
// Uncomment the following line to generate a named log file  
// -LOGFILE ncvlog.log  
// Comment the following line to force recompilation.  
-UPDATE  
  
// Source Files  
.top.v  
  
// Library Files and Directories  
.libs/comb_logic.v  
.models/or2.v  
.models/and2.v
```

### ***ncprep* Output File—ncelab.args**

*ncprep* creates the ncelab.args file. This file is the argument file for the ncelab command. See [Chapter 8, “Elaborating the Design with ncelab.”](#) for details on *ncelab*.

```
// ncelab.args: Arguments passed to the NC-Verilog elaborator.  
// Created by nprep on Thu May 13 12:53:53 1999  
-ERRORMAX 25  
  
// Frequently occurring Verilog Warnings.  
-NOWARN CUVWSP  
-NOWARN CUVWSP  
  
// Turn on informative messages.  
-MESSAGES  
// -NOCOPYRIGHT  
  
// Uncomment the following line to generate a named log file  
// -LOGFILE ncelab.log  
  
-LIBVERBOSE  
-MAXDELAYS  
-NOTIMINGCHECKS  
  
// Translated Options.  
+libverbose  
+gui  
+max_err_count+25  
+notimingchecks  
+maxdelays  
+nowarnTFNPC  
+nowarnTFMPS  
  
// Top level module(s)  
top  
  
// Design snapshot name  
-SNAPSHOT worklib.top:v
```

### ***ncprep* Output File—*ncsim.args***

*ncprep* creates the *ncsim.args* file. This file is the argument file for the *ncsim* command. See [Chapter 9, “Simulating Your Design with \*ncsim\*,”](#) for details on *ncsim*.

```
// ncsim.args: Arguments passed to the NC-Verilog simulator.  
// Created by ncpred on Thu May 13 12:53:53 1999  
-ERRORMAX 25  
  
// Frequently occurring Verilog Warnings.  
  
// Turn on informative messages.  
-MESSAGES  
// -NOCOPYRIGHT  
  
// Uncomment the following line to generate a named log file  
// -LOGFILE ncsim.log  
  
// This will bring you to an interactive TCL session.  
-TCL  
-GUI  
  
// Translated Options.  
+libverbose  
+gui  
+max_err_count+25  
+notimingchecks  
+maxdelays  
+nowarnTFNPC  
+nowarnTFMPS  
  
// Design snapshot name  
worklib.top:v
```

## **Verilog-XL Command-Line Options Translation**

*ncprep* translates all applicable Verilog-XL command-line options into options for *ncvlog*, *ncelab*, and *ncsim*. The tables in this section detail which XL options are translated into which NC-Verilog simulator options.

### **Verilog-XL Dash (-) Option Translation Table**

The *ncprep* utility evaluates Verilog-XL command line *dash* options and generates the equivalent NC-Verilog simulator options. The following table lists the translations that *ncprep* performs.

**Table 18-1 Dash (-) Option Translation**

<b>Verilog-XL Dash (-) Option</b>	<b>Action</b>
-a	Ignored.
-c	Ignored.
-d	Ignored.
-f <i>filename</i>	Translates the Verilog-XL command line options contained in <i>filename</i> .
-i <i>filename</i>	Ignored because NC-Verilog cannot use a file of Verilog commands. Prints a message telling you to use the +tcl+ option to include a file of Tcl commands.
-k <i>filename</i>	Translated to <i>ncsim -keyfile</i> .
-l <i>filename</i>	<i>ncprep</i> generates a log file called <i>ncprep.log</i> by default. Use the <i>ncprep -l filename</i> option to give this log file a different name.
-q	Comments out the -messages option and inserts the -nocopyright option in the <i>ncvlog</i> , <i>ncelab</i> , and <i>ncsim</i> arguments files.
-r	Ignored.
-s	Translated to <i>ncsim -tcl</i> .
-t	Ignored.

## NC-Verilog Simulator Help

### Utilities

---

**Table 18-1 Dash (-) Option Translation**

<b>Verilog-XL Dash (-) Option</b>	<b>Action</b>
<code>-u</code>	Translated to <code>ncvlog -upcase</code> .
<code>-v library</code>	Defines <i>library</i> in the <code>cds.lib</code> file and sets the <code>LIB_MAP</code> variable in the <code>hdl.var</code> file.
<code>-w</code>	Translated to <code>ncvlog -neverwarn</code> , <code>ncelab -neverwarn</code> , and <code>ncsim -neverwarn</code> .
<code>-x</code>	Ignored. Prints a message telling you that expansion has a severe performance impact and is almost never necessary in NC-Verilog. Use <code>+expand</code> to force expansion of all vectors.
<code>-y library</code>	Defines <i>library</i> in the <code>cds.lib</code> file and sets the <code>LIB_MAP</code> variable in the <code>hdl.var</code> file.

### **Verilog-XL Plus (+) Option Translation Table**

The following table shows which Verilog-XL command-line plus options are translated by `ncprep`. There are many other Verilog-XL plus options documented in the Verilog-XL Reference Manual that have no counterparts in the NC-Verilog simulator or that are ignored with a warning.

**Table 18-2 Plus (+) Option Translation**

<b>Verilog-XL Plus (+) Option</b>	<b>Action</b>
<code>+define+arg</code>	<code>ncvlog <a href="#">-define</a></code>
<code>+delay_mode_distributed</code>	<code>ncelab <a href="#">-delay_mode</a> distributed</code>
<code>+delay_mode_path</code>	<code>ncelab <a href="#">-delay_mode</a> path</code>
<code>+delay_mode_unit</code>	<code>ncelab <a href="#">-delay_mode</a> unit</code>
<code>+delay_mode_zero</code>	<code>ncelab <a href="#">-delay_mode</a> zero</code>
<code>+gui</code>	<code>ncsim <a href="#">-gui</a></code>
<code>+inmdir+arg</code>	<code>ncvlog <a href="#">-inmdir</a></code>
<code>+libext+arg</code>	Sets the <code>VIEW_MAP</code> variable in the <code>hdl.var</code> file.

## NC-Verilog Simulator Help

### Utilities

---

**Table 18-2 Plus (+) Option Translation**

<b>Verilog-XL Plus (+) Option</b>	<b>Action</b>
+libverbose	ncelab <a href="#">-libverbose</a>
+licq*	ncsim <a href="#">-licqueue</a>
+loadplil=arg	ncelab <a href="#">-loadplil</a>
+loadvpi=arg	ncelab <a href="#">-loadvpi</a>
+maxdelays	ncelab <a href="#">-maxdelays</a>
+max_err_count+arg or +max_error_count+arg	ncvlog, ncelab, and ncsim <a href="#">-errormax</a>
+mindelays	ncelab <a href="#">-mindelays</a>
+multisource_int_delays	ncelab <a href="#">-intermod_path</a>
+neg_tchk	ncelab <a href="#">-neg_tchk</a>
+no_notifier	ncelab <a href="#">-nonotifier</a>
+no_pulse_msg	ncsim <a href="#">-epulse no msg</a>
+nosdfwarn	ncelab <a href="#">-sdf_no_warnings</a>
+no_show_cancelled_e or +noshow_cancelled_e	ncelab <a href="#">-epulse noneq</a>
+no_tchk_msg or +notchkmsg	ncelab <a href="#">-no_tchk msg</a>
+notimingchecks	ncelab <a href="#">-notimingchecks</a>
+nowarn+arg	ncvlog, ncelab, and ncsim <a href="#">-nowarn</a>
+ntc_warn	ncelab <a href="#">-ntc_warn</a>
+pathpulse	ncelab <a href="#">-pathpulse</a>
+profile	ncsim <a href="#">-profile</a>
+pulse_e/arg	ncelab <a href="#">-pulse_e</a>

**Table 18-2 Plus (+) Option Translation**

Verilog-XL Plus (+) Option	Action
+pulse_e_style_ondetect	ncelab <a href="#">-epulse_ondetect</a>
+pulse_e_style_onevent	ncelab <a href="#">-epulse_onevent</a>
+pulse_int_e/arg	ncelab <a href="#">-pulse_int_e</a>
+pulse_int_r/arg	ncelab <a href="#">-pulse_int_r</a>
+pulse_r/arg	ncelab <a href="#">-pulse_r</a>
+sdf_nocheck_celltype	ncelab <a href="#">-sdf_nocheck_celltype</a>
+sdf_no_warnings or +sdf_nowarnings	ncelab <a href="#">-sdf_no_warnings</a>
+sdf_verbose	ncelab <a href="#">-sdf_verbose</a>
+show_cancelled_e	ncelab <a href="#">-epulse_neg</a>
+transport_int_delays	ncelab <a href="#">-intermod_path</a>
+turbo3	Ignored. The simulator runs in performance mode by default.
+typdelays	ncelab <a href="#">-typdelays</a>
+venv	ncsim <a href="#">-gui</a>
+vcw	ncsim <a href="#">-gui</a>

**Note:** All other Verilog-XL plus options are ignored.

## Using Interactive Debugging Commands

The NC-Verilog simulator interactive debugging commands are based on the Tool Command Language (Tcl). Using Tcl instead of the Verilog HDL gives the simulator the flexibility to display and manipulate mixed-language constructs.

To use interactive debugging commands, halt the simulator by using the `-tcl` option when you invoke `ncsim`, the `$stop` system task, or by pressing Control-C (or the host operating system's interrupt key). When you halt the simulator, the `ncsim>` prompt appears, and you can then enter commands.

## NC-Verilog Simulator Help

### Utilities

---

For more information on the interactive commands, type `help` at the `ncsim>` prompt. See [Chapter 12, “Using the Tcl Command-Line Interface,”](#) for detailed information on each interactive command.

With both Verilog-XL and the NC-Verilog simulator you can use the SimVision analysis environment to debug your design. With Verilog-XL, you invoke the environment with the `+gui` option. With the NC-Verilog simulator, use the `-gui` option when you invoke the simulator (`ncsim`). For example:

```
% ncsim -gui top
```

The following table lists common Verilog-XL commands and their NC-Verilog simulator equivalents.

Verilog-XL	NC-Verilog	Description
<code>'define</code>	<code>alias</code>	Alias a command
<code>.</code>	<code>run</code>	Continue with the simulation
<code>;</code>	<code>run -step</code>	Step a single statement
<code>,</code>		Step and trace a single statement
<code>:</code>		Print current location
<code>&lt;NUMBER&gt;</code>	<code>!&lt;number&gt;</code>	Re-execute a previous command
<code>-&lt;NUMBER&gt;</code>		Disable a previous command
<code>\$history</code>	<code>history</code>	Show a history of previous commands
<code>\$showscopes;</code>	<code>scope -show</code>	Show current scope and subsscopes
<code>\$list;</code>	<code>scope -list</code>	List source HDL of current scope
<code>\$showvars;</code>	<code>value&lt;object&gt;</code>	Show value of object
	<code>time</code>	Show current simulation time
<code>#10 \$stop; .</code>	<code>run 10&lt;ns&gt;</code>	Run for some amount of time then stop
<code>\$finish (0,1,2)</code>	<code>finish&lt;0,1,2&gt;</code>	Finish the simulation and exit
	<code>exit</code>	

## PLI Tasks

Designs that use PLI or VPI routines (referred to as *user system tasks*) must be compiled and linked into a dynamic library or linked statically with the NC-Verilog simulator tools. When *ncprep* detects a user system task, it prints a message informing you that the NC-Verilog simulator will need to be specially set up to run with your PLI or VPI code.

See the [PLI](#) and [VPI](#) reference manuals for details on how to link system tasks to the NC-Verilog simulator tools.

## SDF Annotation

You can annotate timing check and delay data contained in an SDF file. The NC-Verilog simulator supports SDF versions 1.0, 2.0, 2.1, and 3.0. For versions 2.0 and above, use the SDFVERSION statement in the header of the SDF file to specify the version.

SDF annotation is performed during elaboration. The *ncprep* utility detects the presence of \$sdf\_annotation system tasks in the design source files and if the \$sdf\_annotation system tasks are scheduled to run at time 0 and meet other requirements, annotation is performed automatically when you run the RUN\_NC script.

See “[\\$sdf\\_annotation System Task](#)” on page 803 for a description of the \$sdf\_annotation system task. See “[Requirements for \\$sdf\\_annotation System Tasks](#)” on page 811 for a description of the rules that apply to the \$sdf\_annotation tasks for automatic SDF annotation.

The NC-Verilog simulator reads only compiled SDF files. You can specify the name of the SDF source file or the name of the compiled SDF file as an argument in a \$sdf\_annotation task.

If the elaborator determines that the \$sdf\_annotation argument is a text SDF file, it then looks for a corresponding compiled file (*sdf\_filename.x*). For example, if the SDF file is *cpu.sdf*, the elaborator looks for *cpu.sdf.x*.

- If the elaborator doesn't find a corresponding compiled file, the elaborator issues a warning message and then spawns the *ncsdfc* utility to automatically compile the SDF file. The compiled SDF file is written to the directory that contains the SDF file.
- If a compiled file exists, *ncsdfc* checks to make sure that the date of the compiled file is newer than the date of the source SDF file and that the version of the compiled file matches the version of *ncsdfc*. If either check fails, the SDF file is recompiled. Otherwise, the compiled file is simply read.

Log and error messages generated by the elaborator and by *ncsdfc* while compiling the SDF file specified in a `$sdf_annotate()` system task are contained in `ncelab.log`.

Annotation-specific messages from the SDF annotator are sent to the screen and to the log file specified by the log file argument of the `$sdf_annotate()` system task.

You can run *ncsdfc* to compile the SDF file and then use the name of the compiled file as the argument to `$sdf_annotate`. This allows you to, for example, use an old compiled file even if a new SDF source file exists. For example, you could run *ncsdfc* to compile `cpu.sdf`, using the `-output` option to generate a compiled file called `cpu_preroute.sdf`, and then use `cpu_preroute.sdf` as the argument to `$sdf_annotate`. When you elaborate, *ncsdfc* checks that the version of the compiled file matches the version of *ncsdfc*. If it does, the file is read. If it doesn't, the `$sdf_annotate` task is ignored with a warning.

See “[ncsdfc](#)” on page 906 for details on compiling SDF files with *ncsdfc*.

If a `$sdf_annotate` task violates the requirements listed in “[Requirements for \\$sdf\\_annotate System Tasks](#)” on page 811, the elaborator generates warning messages telling you that the task will be ignored. The following example illustrates the warning messages generated if a `$sdf_annotate` task is preceded by a delay statement.

```
ncelab: *W,CUSDEC: A Delay or an Event Control was found before the SDF System Task..  
$sdf_annotate("my.sdf");  
|  
ncelab: *W,CUSSTI (./top.v,11|12): This SDF System Task will be Ignored..  
$sdf_annotate("my.sdf");
```

It is possible to override the default automatic SDF annotation mechanism and force annotation. To do this, edit the `ncelab.args` file and insert the `-sdf_cmd_file filename` option. See “[Using the -sdf cmd file Option](#)” on page 885 for more information.

## Using `$test$plusargs` to Selectively Perform Annotations

You can use an `if($test$plusargs("string"))` construct to selectively perform the annotations.

In the following example, `$test$plusargs` is used to check for the presence of plus options on the command line. If `+preroute` appears on the `ncprep` command line, the file `preroute.sdf` is used for annotation. If `+postroute` appears on the command line, the file `postroute.sdf` is used.

```
module top;
    ...
    circuit m1(i1,i2,i3,o1,o2,o3);
    initial
        if ($test$plusargs("preroute"))
            $sdf_annotate("preroute.sdf", m1);
        else
            if ($test$plusargs("postroute"))
                $sdf_annotate("postroute.sdf", m1);
            else
                $display("No SDF annotation being done for this run");
        //stimulus and response checking
        ...
    endmodule
```

## Specifying Precision

The *ncsdfc* utility always compiles the SDF file with a precision of 1 fs. The elaborator annotates each module using the precision of the module or the precision set by using the *ncelab -sdf\_precision command\_line* option.

## Turning Off SDF Annotation

To specify that you don't want to perform SDF annotation, you can:

- Insert the `-noautosdf` option in the *ncelab.args* file before you execute the *RUN\_NC* script.
- Comment out the `$sdf_annotate` system task(s) in the Verilog source file.

## Using the `-sdf_cmd_file` Option

If the `$sdf_annotate` system tasks in the design do not meet the requirements listed in “[Requirements for \\$sdf\\_annotate System Tasks](#)” on page 811, you can override the automatic annotation mechanism and force the annotation. To do this:

- Compile the SDF file with *ncsdfc*. For example,  
`% ncsdfc my.sdf`
- Write an SDF command file.

See “[Writing an SDF Command File](#)” on page 791 for details on the SDF command file.

## NC-Verilog Simulator Help

### Utilities

---

If you have run *ncprep* and have executed the `RUN_NC` script, the elaborator generates warning messages if the `$sdf_annotate` system tasks in the design do not meet the requirements for automatic SDF annotation. These messages include a warning message like the following example. Use the information in the warning message to create the SDF command file.

```
ncelab: *W,CUSDFI (./top.v,11|12): To force annotation, use option  
-SDF_CMD_FILE <cmd_file_name>.
```

```
// SDF Command file  
// Cut and Paste the next set of lines into a SDF command file if it is necessary  
// to force annotation. Please remember to compile my.sdf before using it.  
COMPILED_SDF_FILE = "my.sdf.X",  
SCOPE = top,  
LOG_FILE = "sdf1.log";
```

**Note:** At the time when the elaborator generates these warning messages, the scope argument to the `$sdf_annotate` task (the second argument) cannot be fully resolved. After cutting and pasting the line(s) in the warning message into an SDF command file, check the command statements and modify any `SCOPE` statements, if necessary, so that the proper scope is specified for annotation.

- Edit the `ncelab.args` file to include the `-sdf_cmd_file filename` option to include the SDF command file.
- Execute the `RUN_NC` script.

## Troubleshooting

This section describes common problems that you may encounter when you convert a design to NC-Verilog. In each case, manual intervention in the form of editing the scripts or modifying the Verilog source, will be required.

The following problem conditions are illustrated in this section:

- Duplicate Modules
- Scanning Multiple -v Files
- Race Conditions
- Renaming Modules

### Duplicate Modules

Having more than one module with the same name in the set of source files may cause unpredictable differences in the behavior of the NC-Verilog simulator as compared to Verilog-XL.

A number of duplicate module scenarios and solutions are presented here:

**Problem:** A `-v` file contains multiple modules with the same name.

If a `-v` file contains multiple modules of the same name, Verilog-XL detects and compiles only the first one it encounters. Subsequent modules of the same name are never used. *ncvlog* detects this and displays a warning message that a module with the same lib, cell, and view name is being recompiled.

**Solution:** Remove the second module or use the compiler directive `'ifdef`

In order to correct this problem, you must remove the second module or use the conditional compiler directive `'ifdef` to conditionally exclude the second module during compilation. Then rerun *ncvlog* or the `RUN_NC` script.

The following example shows a file named `multimod.v` that contains multiple modules with the same name. *ncvlog* detects the duplicate module and displays a warning message.

## NC-Verilog Simulator Help

### Utilities

---

```
% more multimod.v
// First module.
module mod();
    initial
        $display ("I am mod #1");
endmodule

// Second module.
module mod();
    initial
        $display ("I am mod #2");
endmodule

% more top.v
module top();
    mod mod_inst();
endmodule

% more argfile
top.v
-v multimod.v

// Run Verilog-XL.
% verilog -f argfile
VERILOG-XL 2.4.2  Oct 18, 1996  13:31:11
...
...
...
Highest level modules:
top
// Verilog-XL executes the $display statement from the first module.
I am mod #1
0 simulation events (use +profile or +listcounts option to count)
CPU time: 0.6 secs to compile + 0.2 secs to link + 0.0 secs in simulation
End of VERILOG-XL 2.4.2  Oct 18, 1996  13:31:14

// Run ncprep.
% ncprep -f argfile
ncprep: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
...
...
...
```

## NC-Verilog Simulator Help

### Utilities

---

```
ncprep: v1.0.(p1) Exiting on Oct 17, 1996 10:15:58
// Run the RUN_NC script.
% RUN_NC
ncvlog: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
file: top.v
    module worklib.top
        errors: 0, warnings: 0
file: multimod.v
    module multimod.mod
        errors: 0, warnings: 0
// ncvlog detects the duplicate module.
module mod();
|
ncvlog: *W,RECOMP (multimod.v,6|9): recompiling module/udp multimod.mod.
First compiled from line 1 of multimod.v.
module multimod.mod
    errors: 0, warnings: 1
ncelab: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
Elaborating the design hierarchy:
...
...
...
Writing initial simulation snapshot: worklib.top:snap
ncsim: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
Loading snapshot worklib.top:snap ..... Done
ncsim> source /usr2/cds/hpux/inca/tools/inca/files/ncsimrc
ncsim> run
// ncsim executes the $display statement from the second module.
I am mod #2
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
%
```

## Duplicate Modules (continued)

**Problem:** A `-v` or `-y` file contains a module with the same name as one of the top level modules.

If a `-v` or `-y` file contains a module with the same name as one of the top level modules, `ncelab` displays an error message indicating an ambiguous top level module.

**Solution:** In the `ncelab.args` file, specify exactly which module to use.

To correct an ambiguous top level module, you must edit the `ncelab.args` file to specify exactly which module to use. You can do this by adding the name of the library and/or the name of the view to the top level module specification of the `ncelab.args` file.

The following example shows a `-v` file containing a module with the same name as the top level module and the entry in the `ncelab.args` file that corrects the problem.

```
% more multimod.v
// There are two modules named top.
module top();
    initial
        $display ("I am named top");
endmodule
module mod2();
    initial
        $display ("I am named mod2");
endmodule

% more top.v
module top();
    mod2 mod_inst();
endmodule

% more argfile
top.v
-v multimod.v
%
// Run Verilog-XL.
% verilog -f argfile
VERILOG-XL 2.4.2  Oct 18, 1996 13:48:19
...
...
...
```

## NC-Verilog Simulator Help

### Utilities

---

Highest level modules:

```
top
// Verilog-XL prints the $display statement from the second module.
I am named mod2
0 simulation events (use +profile or +listcounts option to count)
CPU time: 0.8 secs to compile + 0.2 secs to link + 0.1 secs in simulation
End of VERILOG-XL 2.4.2 Oct 18, 1996 13:48:22
// Run ncprep.
% ncprep -f argfile
ncprep: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
...
...
...
Translation successful.
ncprep: v1.0.(p1) Exiting on Oct 17, 1996 16:40:23
// Execute the RUN_NC script.
% RUN_NC
ncvlog: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
file: top.v
...
...
...
ncelab: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
// ncelab detects the duplicate module and issues an error message.
ncelab: *E,MTOMDU: More than one unit matches 'top':
        module multitop.top:module (VST)
        module worklib.top:module (VST).
ncsim: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
ncsim: *F,NOSNAP: snapshot 'worklib.top:snap' does not exist in the libraries.
%
```

## Duplicate Modules (continued)

**Problem:** Multiple `-v` and `-y` files contain multiple modules with the same name.

If more than one `-y` or `-v` file contains the same module name, *ncelab* may, in rare cases, perform the instance binding in a different order than Verilog-XL.

**Solution:** This case can only be detected when using the `libverbose` features of Verilog-XL and the NC-Verilog simulator, or by comparing simulation results and noting differences.

The following example shows more than one `-v` file containing the same module name and the output of *ncelab* with the `-libverbose` option.

```
% more top.v
module top();
    a a1();
    b b1();
endmodule

% more one.v
// First module named b.
module b();
    initial
        $display ("I am B in one.v %m");
endmodule

% more two.v
module a();
    initial
        $display ("I am A in two.v %m");
    b b2();
endmodule

// Second module named b.
module b();
    initial
        $display ("I am B in two.v %m");
endmodule

% more argfile
top.v
-v a.v
-v b.v
```

## NC-Verilog Simulator Help

### Utilities

---

```
+libverbose
// Run Verilog-XL
% verilog -f argfile
VERILOG-XL 2.4.2  Oct 18, 1996  13:57:15
...
...
...
Compiling source file "top.v"
Scanning library file "one.v"
Compiling library module (b)
Scanning library file "two.v"
Compiling library module (a)
Highest level modules:
top
// Output from Verilog-XL
I am A in two.v top.a1
I am B in one.v top.a1.b2
I am B in one.v top.b1      // This output differs from NC-Verilog
0 simulation events (use +profile or +listcounts option to count)
CPU time: 0.8 secs to compile + 0.0 secs to link + 0.1 secs in simulation
End of VERILOG-XL 2.4.2  Oct 18, 1996  14:10:11
// Run ncprep
% ncprep -f argfile
ncprep: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
...
...
...
Translation successful.
ncprep: v1.0.(p1) Exiting on Oct 17, 1996  16:40:23
// Execute the RUN_NC script
% RUN_NC
ncvlog: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
...
...
...
ncelab: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
...
...
...
Writing initial simulation snapshot: worklib.top:snap
ncsim: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
```

## NC-Verilog Simulator Help

### Utilities

---

```
Loading snapshot worklib.top:snap ..... Done
ncsim> source /usr2/cds/hpux/inca/tools/inca/files/ncsimrc
ncsim> run
// Output from ncsim. Note how the output differs from Verilog-XL.
I am A in two.v top.a1
I am B in two.v top.a1.b2
I am B in two.v top.b1
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
%
```

### Scanning Multiple -v Files

**Problem:** Verilog-XL may scan `-v` files more than once while searching for modules. Compiler directives and macros that resolve to different values during each scan may cause unpredictable results.

**Solution:** The *ncprep* utility ensures that each source file is parsed only once and has only one set of cross-file inherited macros and directives.

The following example shows a `'timescale` directive that changes from one scan of the file to the next.

```
% more top.v
`timescale 1ns/1ns      // `timescale directive
module top();
    a a1();
endmodule

% more file1.v
module a();
    initial
        #1 $display ("%t I am A",$time);
        b b1();
endmodule
module c();
    initial
        #1 $display("%t I am C",$time);
endmodule
```

## NC-Verilog Simulator Help

### Utilities

---

```
% more file2.v
`timescale 1ps/1ps          // 'timescale directive changes from one scan of the
                           // file to another.

module b();
    initial
        #1 $display("%t I am B", $time);
        c c1();
endmodule

% more argfile
top.v
-v
file1.v
-v
file2.v
// Run Verilog-XL
% verilog -f argfile
VERILOG-XL 2.4.2  Oct 18, 1996  13:57:15
...
...
...
// The 'timescale directive has changed between the first and last scan of
// file1.v.
Compiling source file "top.v"
Scanning library file "file1.v"
Scanning library file "file1.v"
Scanning library file "file2.v"
Scanning library file "file2.v"
Scanning library file "file1.v"
Highest level modules:
top
      1 I am B
      1 I am C
      1000 I am A
0 simulation events (use +profile or +listcounts option to count)
CPU time: 0.6 secs to compile + 0.1 secs to link + 0.0 secs in simulation
End of VERILOG-XL 2.4.2  Oct 18, 1996  16:25:25
// Run ncprep
% ncprep -f argfile
ncprep: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
...
...
```

## NC-Verilog Simulator Help

### Utilities

---

```
...
Translation successful.
ncprep: v1.0.(p1) Exiting on Oct 18, 1996 16:33:18
// Execute the RUN_NC script
% RUN_NC
ncvlog: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
...
...
...
ncelab: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
...
...
...
Writing initial simulation snapshot: worklib.top:snap
ncsim: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
Loading snapshot worklib.top:snap ..... Done
ncsim> source /usr2/cds/hpux/inca/tools/inca/files/ncsimrc
ncsim> run
      1 I am B // Note how the simulation times differ from Verilog-XL.
      1000 I am A
      1000 I am C
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

## Race Conditions

**Problem:** Multiple simulation events that occur at the same time may cause a race condition. In the case of a race condition, Verilog-XL and the NC-Verilog simulator will have different simulation results. Detecting a race condition is a complex problem that can not be detected by *ncprep*.

**Solution:** Race conditions can only be detected when you compare simulation results, and noting differences, trace the cause back to multiple simultaneous events.

The following example illustrates a race condition that can only be detected when you compare the simulation results.

## NC-Verilog Simulator Help

### Utilities

---

```
% more race.v
module top();
    reg r1, r2;
    xor x1(o, i1, r2);
    buf #1 b1(i1, r1), b2(i2, r2);
    initial
        begin
            r1 = 1;
            r2 = 0;
        end
// Multiple simultaneous events may cause race conditions.
    always @(r1) r1 <= #1 ~r1;
    always @(r2) r2 <= #2 ~r2;

    always @() $display ("time %t o = %b", $time, o);
    always @() $display ("time %t r1 = %b", $time, r1);
    always @() $display ("time %t r2 = %b", $time, r2);

    initial
        #100 $finish;
endmodule
// Run Verilog-XL
% verilog race.v
VERILOG-XL 2.4.2  Oct 24, 1996  14:21:28
...
...
...
Compiling source file "race.v"
Highest level modules:
// Output from Verilog-XL
top
time      1  r2 = 0
time      1  r1 = 1
time      1  o = 1
L20 "file.v": $finish at simulation time 100
0 simulation events (use +profile or +listcounts option to count) + 5 accelerated
events
CPU time: 0.8 secs to compile + 0.3 secs to link + 0.0 secs in simulation
End of VERILOG-XL 2.4.2  Oct 24, 1996  13:14:29
```

## NC-Verilog Simulator Help

### Utilities

---

```
// Run ncprep
% ncprep race.v
ncprep: v1.0.(p1): (c) Copyright 1995, 1996 Cadence Design Systems, Inc.
...
...
...
Translation successful.
ncprep: v1.0.(p1) Exiting on Oct 24, 1996 14:25:33
// Execute the RUN_NC script
% RUN_NC
ncvlog: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
...
...
...
ncelab: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
...
...
...
ncsim: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
Loading snapshot worklib.top:snap ..... Done
ncsim> source /usr2/cds/hpux/inca/tools/inca/files/ncsimrc
ncsim> run
// Output from NC-Verilog. Note that the event ordering of the NC-Verilog output
// is reversed from the Verilog-XL output. This race condition can only be
// detected by comparing the output results.
time      1  o = 1
time      1  r1 = 1
time      1  r2 = 0
Simulation complete via $finish(1) at time 100 NS + 0
./file.v:20 #100 $finish;
ncsim> exit
```

## Renaming Modules

**Problem:** When more than one module of the same name is instantiated in a design, Verilog-XL may rename one of the instantiated modules. The NC-Verilog simulator uses a unique lib, cell, and view name and will not rename modules. *ncvlog* or *ncelab* will detect the naming inconsistency and display an error message.

**Solution:** Rename one of the modules.

The following example shows more than one module of the same name instantiated in a design.

```
% more file1.v
module top();
    a a1();           // An instantiation of multiple modules with the
                      // same name.
    b b1();
endmodule

module a();
    initial
        $display ("I am mod a in File1.top");
endmodule

% more b.v
module b();
endmodule

module a();
    initial
        $display ("I am mod a in b.v");
endmodule

// Run Verilog-XL
% verilog file1.v -y . +libext+.v
VERILOG-XL 2.4.2   Oct 23, 1996  14:11:10
...
...
...
Highest level modules:
top
a$b$1           // The module named top has been renamed.
```

## NC-Verilog Simulator Help

### Utilities

---

```
I am mod a in File1.top
I am mod a in b.v
0 simulation events (use +profile or +listcounts option to count)
CPU time: 0.9 secs to compile + 0.2 secs to link + 0.0 secs in simulation
// Run ncprep
% ncprep file1.v -y . +libext+.v
ncprep: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
...
...
...
Translation successful.
// Execute the RUN_NC script
% RUN_NC
ncvlog: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
file: file1.v
    module local.top:v
        errors: 0, warnings: 0
    module local.a:v
        errors: 0, warnings: 0
file: ./b.v
    module local.b:v
        errors: 0, warnings: 0
module a();
|
ncvlog: *W,RECOMP (./b.v,4|7): recompiling module/udp local.a:v.
First compiled from line 6 of file1.v.
    module local.a:v
        errors: 0, warnings: 1
ncelab: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
// Verilog-XL has renamed the duplicate module and ncelab detects the error.
ncelab: *E,NOUNIT: Unable to find a unit named 'a$b$1' in the libraries.
ncsim: v1.0.(p1): (c) Copyright 1995,1996 Cadence Design Systems, Inc.
ncsim: *F,NOSNAP: snapshot 'worklib.top:snap' does not exist in the libraries.
```

## **ncrm**

The *ncrm* utility lets you delete the contents of a library. You can delete the entire contents of a library or you can selectively delete specific design units: cells, views, and snapshots.

Syntax:

```
ncrm [-options] {[lib.]cell[:view]} ...
      [-library library_name] ...
      [-snapshot snapshot] ...}
```

Examples:

```
% ncrm top
% ncrm top:module
% ncrm worklib.top:module
% ncrm -library worklib
% ncrm -snapshot top
```

## **ncrm Command Syntax**

Invoke *ncrm* with options and arguments. Options can occur in any order. Parameters to options must immediately follow the option they modify.

```
ncrm [-options] {
      {[lib.]cell[:view]} ...
      [-library library_name] ...
      [-snapshot snapshot] ...
}

[-Append log]
[-Cdslib filename]
[-Force]
[-HDLvar filename]
[-Help]
[-Library library_name]
[-LogFile logfile_name]
[-Messages]
[-NCError warning_code]
[-NCFatal {warning_code | error_code}]
[-NEverwarn]
[-NOCopyright]
[-NOLog]
```

[ -NOSTdout ]  
[ -NOWarn *warning\_code* ]  
[ -Snapshot *snapshot* ]  
[ -Version ]

## **ncrm Command Options**

The following list describes the options you can use with the `ncrm` command. Options can be entered in upper or lower case. In the list, capital letters indicate the shortest possible abbreviation for an option.

### **-Append\_log**

Appends log information from multiple runs of *ncrm* to one log file. This option is overridden by the `-nolog` option.

### **-Cdslib *filename***

Uses the specified `cds.lib` file. See “[The cds.lib File](#)” on page 110 for more information.

### **-Force**

Disables printing of all error messages.

### **-HDIvar *filename***

Uses the specified `hdl.var` file. See “[The hdl.var File](#)” on page 118 for more information.

### **-HElp**

Prints a brief summary of the `ncrm` command options. No other action besides printing the help message takes place.

### **-Library *library\_name***

Removes all elements in the specified library.

**-Logfile *logfile\_name***

Uses the specified name for the log file instead of the default name `ncrm.log`.

**Note:** Information is only written to the log file when you use the `-messages` option.

**-Messages**

Prints informative messages during execution.

**Note:** Information is only written to the log file when you use the `-messages` option.

**-NCError *warning\_code***

Increase the severity level of the specified warning message from warning to error. The `warning_code` argument is the message code (mnemonic) that appears in the warning message following the severity code.

Example:

```
% ncrm -messages -library worklib -ncerror ABCDEF
```

You can include multiple `-ncerror` options on the command line.

Using this option can change the behavior of the tool because functions that return errors instead of warnings may behave differently.

**-NCFatal {*warning\_code* | *error\_code*}**

Increase the severity level of the specified warning message or error message from warning or error to fatal. The `warning_code` or `error_code` argument is the message code (mnemonic) that appears in the message following the severity code.

Example:

```
% ncrm -messages -library worklib -ncfatal LMNOPQ
```

You can include multiple `-ncfatal` options on the command line.

**-NEverwarn**

Disables printing of all warnings.

**-NOCopyright**

Suppresses printing the copyright banner.

**-NOLog**

Does not generate a log file. By default, *ncrm* generates a log file called `ncrm.log`. This option is overridden by the `-logfile` option.

**-NOStdout**

Suppresses printing to the screen.

**-NOWarn *warning\_code***

Disables the printing of the specified warning. The *warning\_code* is the sequence of characters following the error severity code.

**-Snapshot *snapshot***

Deletes the specified design unit, which is a snapshot.

**-Version**

Prints the version of *ncrm* and exits.

**Example ncrm Command Lines**

The following command deletes all intermediate objects for design unit `top` from the library database file.

```
% ncrm -messages top
```

The following command deletes all intermediate objects for design unit `top:viewa`.

```
% ncrm -messages top:viewa
```

The following command deletes the contents of the library named `worklib`.

```
% ncrm -messages -library worklib
```

## **NC-Verilog Simulator Help**

### Utilities

---

The following command deletes the snapshot for design unit top.

```
% ncrm -messages -snapshot top
```

The following command deletes the snapshot top, everything in library alt\_lib, and all intermediate objects for larry, moe, and curly.

```
% ncrm -mess -snapshot top -library alt_lib larry moe curly
```

The following command deletes the design unit board:behav and sends log information to a file called del.log.

```
% ncrm -messages board:behav -logfile del.log
```

## ncsdfc

The *ncsdfc* utility lets you compile and decompile SDF files.

SDF files must be compiled with *ncsdfc* in order to annotate the timing information contained in the SDF file.

For VHDL VITAL, you must compile the file yourself using *ncsdfc*.

For Verilog, the elaborator automatically invokes *ncsdfc* to compile the SDF file if you are using the `$sdf_annotate` system task to perform the annotation, and if the annotator detects that the file specified as the argument to `$sdf_annotate` is not a compiled SDF file. If you are using an SDF command file, you must invoke *ncsdfc* to compile the SDF file.

See [Chapter 17, “SDF Timing Annotation,”](#) for more information on using an SDF file for timing annotation.

Invoke *ncsdfc* with the `ncsdfc` command. The syntax is as follows:

Syntax:

```
ncsdfc [-options] sdf_filename
```

The `sdf_filename` argument can be the filename of the source SDF file, or an SDF file that has been compressed with the `compress` utility or `gzip`.

**Note:** An SDF file compressed with `gzip` must have a `.gz` suffix.

The output of *ncsdfc* is a compiled SDF file called `sdf_filename.x`. For example, if the name of the SDF file is `dcache.sdf`, the output file is called `dcache.sdf.x`. Use the `-output` option to rename the output file. The output file is placed in the current working directory.

Examples:

```
% ncsdfc dcache.sdf      (Generates dcache.sdf.x)  
  
% gzip -S .gz foo.sdf    (Generates foo.sdf.gz)  
% ncsdfc foo.sdf.gz     (Generates foo.sdf.gz.x)  
  
% ncsdfc dcache.sdf -output mysdf.sdf.X  (Generates mysdf.sdf.X)
```

Use the `-decompile` option to decompile an SDF file if you want to view or edit the file. Decompiling the file is also necessary if you have deleted the original file to save disk space and now want to use the SDF file with another tool. The output file is called `compiled_filename.sdf` by default, and the file is placed in the current working directory.

**Note:** *ncsdfc* saves only delay and timing check values. Decompilation cannot restore, for example, path constraint or waveform data.

## ncsdfc Command Syntax

Invoke *ncsdfc* with options and arguments. Options can occur in any order. Parameters to options must immediately follow the option they modify.

You can specify *ncsdfc* command-line options using the NCSDFCOPTS variable in an *hdl.var* file.

```
ncsdfc [-options] sdf_filename  
  [-Append log]  
  [-CDslib filename]  
  [-COmpile]  
  [-CPutime]  
  [-Decompile]  
  [-HDLvar filename]  
  [-HELP]  
  [-Logfile filename]  
  [-Messages]  
  [-NCError warning_code]  
  [-NCFatal {warning_code | error_code}]  
  [-NEverwarn]  
  [-NOCopyright]  
  [-NOLog]  
  [-NOSTdout]  
  [-Output filename]  
  [-STAtus]  
  [-STDin]  
  [-Update]  
  [-Version]  
  [-Worstcase rounding]
```

## **ncsdfc Command Options**

The following list describes the options you can use with the `ncsdfc` command. Options can be entered in upper or lower case. In the list, capital letters indicate the shortest possible abbreviation for an option.

### **-Append\_log**

Append log information from multiple runs of `ncsdfc` to one log file.

### **-CDslib *filename***

Use the specified `cds.lib` file. See “[The cds.lib File](#)” on page 110 for information on the `cds.lib` file.

### **-CCompile**

Compile the specified SDF file(s). This is the default.

### **-CPutime**

Print CPU time after `ncsdfc` has completed.

### **-Decompile**

Decompile the specified SDF file(s).

### **-HDIvar *filename***

Use the specified `hdl.var` file. See “[The hdl.var File](#)” on page 118 for information on the `hdl.var` file.

### **-HElp**

Print a brief summary of the `ncsdfc` command options. No action takes place other than printing the help message.

**-LogFile *filename***

Use the specified name for the log file instead of the default name `ncsdfc.log`. You can not include this option in an `hdl.var` file. This option overrides the `-nolog` option.

**-Messages**

Print informative messages during execution.

**-NCError *warning\_code***

Increase the severity level of the specified warning message from warning to error. The *warning\_code* argument is the message code (mnemonic) that appears in the warning message following the severity code.

Example:

```
% ncsdfc ibox.sdf -ncerror ABCDEF
```

You can include multiple `-ncerror` options on the command line.

Using this option can change the behavior of the tool because functions that return errors instead of warnings may behave differently.

**-NCFatal {*warning\_code* | *error\_code*}**

Increase the severity level of the specified warning message or error message from warning or error to fatal. The *warning\_code* or *error\_code* argument is the message code (mnemonic) that appears in the message following the severity code.

Example:

```
% ncsdfc ibox.sdf -ncfatal LMNOPQ
```

You can include multiple `-ncfatal` options on the command line.

**-NEverwarn**

Disable printing of all warnings.

**-NOCopyright**

Suppress printing of the copyright banner.

**-NOLog**

Do not generate a log file. By default, *ncsdfc* generates a log file called `ncsdfc.log`. This option is overridden by the `-logfile` option.

**-NOStdout**

Suppress printing of output to the screen.

**-Output *filename***

Redirect output to the specified file.

**-Status**

Display memory and CPU usage statistics.

**-STDin**

Specifies that input is from stdin instead of an ASCII file.

You can use this option to compile a compressed or zipped SDF file. For example,

```
% cat test.sdf.Z | uncompress | ncsdfc -stdin -output test.sdf.X  
% cat test.sdf.gz | gzip -d | ncsdfc -stdin -output test.sdf.gz.X
```

**-Update**

Compile the SDF file only if recompilation is necessary. The SDF file must be recompiled if:

- The SDF source file is newer than the compiled file.
- The SDF source file is not the same file that was used to build the compiled file.
- The version of *ncsdfc* that was used to generate the compiled file is not compatible with the current version of *ncsdfc*.

### **-Version**

Print the version of *ncsdfc* and exit.

### **-Worstcase\_rounding**

For timing values in timing triplets, truncate the min value, round the typ value, and round up the max value. For single values, round the value normally.

## **Example ncsdfc Command Lines**

The following command compiles the SDF file called *ibox.sdf*. The output file is called *ibox.sdf.X*.

```
% ncsdfc ibox.sdf
```

The following command compiles the SDF file called *ibox.sdf*. The *-output* option specifies that the compiled file is to be called *ibox.compiled*.

```
% ncsdfc ibox.sdf -output ibox.compiled
```

The following command compiles a compressed SDF file called *ibox.sdf.gz*. The output file is called *ibox.sdf.gz.X*.

```
% ncsdfc ibox.sdf.gz
```

The following command decompiles *ibox.sdf.X*, a compiled SDF file. The output file is called *ibox.sdf.X.sdfd*.

```
% ncsdfc -decompile ibox.sdf.X
```

The following command decompiles *ibox.sdf.X*, and names the decompiled file *ibox.decompiled*.

```
% ncsdfc -decompile ibox.sdf.X -output ibox.decompiled
```

In the following command, the *-worstcase\_rounding* option specifies that the min value is to be truncated, the typ value rounded, and the max value rounded up.

```
% ncsdfc -worstcase_rounding ibox.sdf
```

For example, the timing values in the following *IOPATH* statement are changed to 0 : .1 : .1.

```
(IOPATH in out (.05:.05:.03))
```

A single timing value is rounded normally. For example, the timing value in the following *IOPATH* statement is changed to 1.

```
(IOPATH in out (.05))
```

## **ncshell**

*ncshell* is a utility that automatically generates shells to facilitate model import.

You can use *ncshell* to generate shells so that you can import:

- VHDL Models into a Verilog simulation
- Verilog Models into a VHDL simulation
- LMSFI Models into a VHDL simulation
- SWIFT Models into a VHDL simulation
- FMI Models into a VHDL simulation

Information on using the *ncshell* utility to import VHDL or Verilog models is contained in [Chapter 10, “Mixed Verilog/VHDL Simulation”](#).

This topic contains the following sections:

- [ncshell Command Syntax](#)
- [ncshell Command Options](#)
- [The Foreign Attribute](#)
- [Importing LMSFI Models into VHDL](#)
- [Importing SWIFT Models into VHDL](#)
- [Importing FMI Models into VHDL](#)

## ncshell Command Syntax

Invoke *ncshell* with options and arguments. Options can occur in any order. Parameters to options must immediately follow the option they modify. Command-line options can be abbreviated to the shortest unique string.

```
ncshell -import {lmsfi | fmi | swift}
           -into {vhdl | verilog} [other_options] argument
```

The *-import* option, which specifies the kind of model that is being imported, and the *-into* option, which specifies the kind of model (that is, the language of the model) into which import is being done, are always required.

The *argument* is one of the following:

- A model name (for (SWIFT and FMI import))
- A file name (for LMSFI import)

The *ncshell* utility has command-line options that are specific to each of the model formats. The options you use depend on the type of model you are importing. The *ncshell* command-line options listed in this section are divided into the following groups:

- General Options
- LMSFI and SWIFT Models Imported into VHDL
- FMI Models Imported into VHDL

### General Options

General options apply to all of the model import formats.

```
[-APPend_log]
[-Errormax integer]
[-FILE filename]
[-Help]
[-IMport {lmsi | fmi | swift}]
[-INTo {vhdl | verilog}]
[-LogFile logfile_name]
[-Messages]
[-NCError warning_code]
[-NCFatal {warning_code | error_code}]
[-NEverwarn]
[-NOCOPYright]
[-NOCOMpile]
```

## NC-Verilog Simulator Help

### Utilities

---

[ -NOLog ]  
[ -NOSTdout ]  
[ -NOWarn *warning\_code* ]  
[ -Shell *shell\_output\_filename* ]  
[ -Version ]

### **LMSFI and SWIFT Models Imported into VHDL**

You can use the following options when the argument to `-import` is `lmsfi` or `swift`.

[ -ALL ]  
[ -Backward ]  
[ -Comp *component\_output\_file* ]

### **FMI Models Imported into VHDL**

You can use the following options when the argument to `-import` is `fmi`.

[ -Backward ]  
[ -FМИLib ]  
[ -FМИEnt ]

## **ncshell Command Options**

This section describes the options that you can use with the `ncshell` command. Options can be entered in upper or lower case. Capital letters indicate the shortest possible abbreviation for an option.

The options listed in this section apply to importing foreign models into both languages (Verilog and VHDL) unless noted.

### **-ALI**

#### **(LMSFI and SWIFT Models imported into VHDL)**

When you import multiple LMSFI and SWIFT models, all the models are captured in a single shell file. You specify the name of the shell with the `-shell` option. Component declarations are also packaged into a single file. You define the name of the component declarations file with the `-comp` option.

**Note:** This option is only valid when the argument to the `-import` option is `lmsfi` or `swift`.

### **-APPend\_log**

Appends log information from multiple *ncshell* runs into one log file. The *-nolog* option overrides this option.

### **-Backward**

Provides compatibility with Leapfrog VHDL and Verilog-XL shells. INCA model shells use a different syntax than that supported by Leapfrog and Verilog-XL. Use this option in the following cases:

- When you import a Verilog-XL shell into an NC-VHDL simulation.
- When you import a Leapfrog VHDL shell into an NC-Verilog simulation.

### **-Comp *output\_filename***

**(All models imported into VHDL)**

Specifies the file name in which a component declaration is written.

Whenever you import a model into VHDL, a component declaration corresponding to the shell is generated (the component declaration is not needed for FMI models and it will not be generated). This option specifies the filename for the generated component declaration. The default filename is *model\_name\_comp.vhd* (the model name with \_comp.vhd appended).

**Note:** This option is applicable only when the argument to *-into* is vhdl.

### **-Errormax *integer***

Specifies the maximum number of errors processed. Aborts *ncshell* after reaching the specified number of errors (expressed as an integer).

### **-Ffile *filename***

Use the command-line arguments contained in the specified file.

You can store frequently used or lengthy command lines by putting command options and arguments in a text file. When you invoke *ncshell* with the *-file* option, the arguments in the specified file are used with the command as if they had been entered on the command line.

Each option, with its arguments must be specified on a separate line.

**-FMIEnt *entity\_name***

**(FMI Model import Only)**

When you import a C interface foreign model into a VHDL simulation, this option specifies the entity name of the foreign model. This option is required when you import an FMI model.

**Note:** This option is only valid when the argument to `-import` is `fmi`.

**-FMILib *library\_name***

**(FMI Model import Only)**

When you import a C interface foreign model into a VHDL simulation, this option specifies the library in which the foreign model resides. This option is required when you import an FMI model.

**Note:** This option is only valid when the argument to `-import` is `fmi`.

**-HElp**

Prints a brief summary of the `ncshell` command options.

**-IMport {lmsfi | fmi | swift}**

Specifies the type of the foreign model you are importing. This option is required for every model import.

**-INto {vhdl | verilog}**

Specifies the generated shell's output HDL format. Use the `vhdl` argument when you want to import a foreign model into a VHDL simulation. Use the `verilog` argument when you want to import a foreign model into a Verilog simulation. This option is required for every model import.

**Note:** When you import FMI, LMSFI, or SWIFT models, the argument to this option must be `vhdl`. The component declaration is not needed for FMI models and it will not be generated.

**-Logfile *logfile\_name***

Specifies the name for the log file. This name is used instead of the default name `ncshell.log`. You can not include this option in an `hdl.var` file. This option overrides the `-nolog` option.

**-Messages**

Displays informational messages during the creation of the shell.

**Note:** Information is only written to the log file when you use the `-messages` option.

**-NCError *warning\_code***

Increase the severity level of the specified warning message from warning to error. The `warning_code` argument is the message code (mnemonic) that appears in the warning message following the severity code.

Example:

```
% ncshell -import lmsfi -into vhdl -all -ncerror ABCDEF
```

You can include multiple `-ncerror` options on the command line.

Using this option can change the behavior of the tool because functions that return errors instead of warnings may behave differently.

**-NCFatal {*warning\_code* | *error\_code*}**

Increase the severity level of the specified warning message or error message from warning or error to fatal. The `warning_code` or `error_code` argument is the message code (mnemonic) that appears in the message following the severity code.

Example:

```
% ncshell -import lmsfi -into vhdl -all -ncfatal LMNOPQ
```

You can include multiple `-ncfatal` options on the command line.

**-NEverwarn**

Disables printing of all warning messages.

**-NOCOPyright**

Suppresses printing of the copyright banner.

**-NOCOMpile**

By default, *ncshell* automatically analyzes the shell it generates. Use this option to override the default analysis.

**-NOLog**

Suppresses creation of the default logfile.

**-NOStdout**

Suppresses output display to screen.

**-NOWarn *warning\_code***

Disables the printing of the specified warning. The *warning\_code* is the sequence of characters following the error severity code.

**-SHell *shellname***

Specifies the name of the generated shell. This option can be used to give the generated shell a name other than the default.

**-Suffix *file\_extension***

**(Models imported into Verilog)**

Specifies a filename extension for a Verilog shell. This option is ignored if you also specify the *-shell* option.

**-View *viewname***

**(Models imported into Verilog)**

Specifies the name of the view that you want the generated shell analyzed into. The default view name is `shell`.

**-VErsion**

Prints the version of `ncshell` and exits.

## The Foreign Attribute

The `ncshell` utility places a `foreign` attribute within the shell to indicate the source format of the foreign model. The `foreign` attribute is created in the architecture of a VHDL model shell and in the module definition of a Verilog shell.

- When you import an LMSFI model into VHDL, the syntax of the `foreign` attribute is:

```
"LMSFI model_name";
```

For example:

```
"LMSFI LMSFI_Model";
```

See "[Importing LMSFI Models into VHDL](#)" on page 920 for more details.

- When you import a SWIFT model into VHDL, the syntax of the `foreign` attribute is:

```
attribute FOREIGN of SmartModel : architecture is "SWIFT model_name";
```

For example:

```
attribute FOREIGN of SmartModel : architecture is "SWIFT tt100";
```

See "[Importing SWIFT Models into VHDL](#)" on page 922 for more details.

- When you import an FMI model into VHDL, the syntax of the `foreign` attribute is:

```
"FMI fmi_library:model_name";
```

For example the following line illustrates the foreign attribute for an FMI model imported into a VHDL shell. The attribute specifies the work library and the model name:

```
"FMI fmi_model_lib:FMI_ALU";
```

See "[Importing FMI Models into VHDL](#)" on page 924 for more details.

## Importing LMSFI Models into VHDL

You should be aware of the following when you import LMSFI models into VHDL.

- The *ncshell* utility assumes that the LMSFI shared library resides at:
  - `$(LM_DIR)/../lib/pa_hp90/libbsfi.sl` for HP platforms
  - `$(LM_DIR)/../lib/sun4.solaris/libbsfi.so` for SUN platforms
- When you use import LMSFI models:
  - You must assign the `LM_LIB` environment variable to the directory where the LMSFI models reside. For example, if you had LMSFI models in the `/usr1/models/lmsfi_models` directory, then you would set the `LM_LIB` environment variable as follows:

```
setenv LM_LIB /usr1/models/lmsfi_models
```
  - If you have LMSFI models that reside in more than one directory, then all the directories must be specified by the `LM_LIB` environment variable (use a colon as the directory separator). For example, if you had LMSFI models in directories with the pathnames `/usr1/mdls/lmsfi_mdls`, and `/usr1/models/old_mdls`, then you would set the `LM_LIB` environment variable as follows:

```
setenv LM_LIB /usr1/mdls/lmsfi_mdls:/usr1/mdls/old_mdls
```
- All LMSFI model filenames must have a `.mdl` extension.
- You can use the `-All` option to import all the models in an LMSFI shared library. The following line is an example:

```
ncshell -import lmsfi -into vhdl -all
```
- *ncshell* analyzes the generated shells in the work directory.

### Example LMSFI Model Imported into VHDL

The following example shows the syntax of the shell and component declaration file that *ncshell* generates from an LMSFI model. The entity-architecture shell and the component declaration are shown below. The `-all` option generates shells for all the LMSFI models in the directories specified by the `LM_LIB` environment variable.

```
ncshell -import lmsfi -into vhdl -all
```

## The LMSFI Model Entity-Architecture Shell

```
library ieee;
use ieee.std_logic_1164.all;
entity LMSFI_model_name is
    generic (
        TimingVersion : string;
        DelayRange : string
    );
    port (      // In an actual shell, these port related labels are replaced
                // with actual values from the LMSFI model.
        port_name : port_mode port_type_with_range := "U...";
    // The number of U's equals the port width.
    );
end LMSFI_model_name;

architecture LogicModel of LMSFI_model_name is
// The foreign attribute
    attribute foreign of LogicModel : architecture is "LMSFI
LMSFI_model_name";
begin
end;
```

**Note:** If you use the -backward option to generate a shell that is compatible with Leapfrog, *ncshell* creates the following foreign attribute in the architecture:

```
attribute foreign of LogicModel : architecture is "LFLM: LFLogicModel";
```

## The LMSFI Model Component Declaration

```
package LogicModels is
component LMSFI_model_name is
    generic (
        TimingVersion : string;
        DelayRange : string
    );
    port (
// In a component declaration, these port related labels are replaced with
// actual values from the LMSFI model.
        port_name : port_mode port_type_with_range_if_any := "U....";
    );
end component;
end LogicModels;
```

## Importing SWIFT Models into VHDL

You need to be aware of the following when you import SWIFT models into VHDL.

- The *ncshell* utility assumes that the SWIFT shared library resides at:
  - `$(LMC_HOME)/.../lib/pa_hp90/libswift.sl` for HP platforms
  - `$(LMC_HOME)/.../lib/sun4.solaris/libswift.so` for SUN platforms
- You can use the `-all` option to import all the models in a SWIFT shared library. The following line is an example:  
`ncshell -import swift -into vhdl -all`
- *ncshell* analyzes the generated shells in the work directory.

## Example SWIFT Model Imported into VHDL

The following example shows the entity-architecture shell and the component declaration that *ncshell* generates from a SWIFT model. You must have the `LMC_HOME` environment variable set to a SWIFT shared library. The `-all` option generates shells for all the SWIFT models in the directories specified by the `LMC_HOME` environment variable.

```
ncshell -import swift -into vhdl -all
```

### The SWIFT Model Entity-Architecture Shell

```
library ieee;
use ieee.std_logic_1164.all;
entity swift_model_name is
// In an actual shell, these generic and port related labels are replaced with
// actual values from the SWIFT model.
    generic (
        generic_name : string;
        --Similar entries for all generics in the model
    );
    port (
        port_name : port_mode std_logic := 'U';
        --Similar entries as above for all non-output ports

        port_name : port_mode std_logic;
        --Similar entries as above for all output ports
    );
end swift_model_name ;
```

## NC-Verilog Simulator Help

### Utilities

---

```
architecture SmartModel of swift_model_name is
    attribute foreign of SmartModel : architecture is "SWIFT swift_model_name";
begin
end;
```

**Note:** If you use the -backward option to generate a shell that is compatible with Leapfrog, *ncshell* creates the following foreign attribute in the architecture:

```
attribute foreign of SmartModel : architecture is "LFSM: LFSmartModel";
```

### The SWIFT Model Component Declaration

```
package SmartModels is
component swift_model_name is
    // In the component declaration, these port and generic related labels are
    // replaced with actual values from the SWIFT model.
    generic (
        generic_name : string;
        --All possible options for the generic value listed as a comment
        --Similar entries for all generics in the model
    );
    port (
        port_name : port_mode std_logic := 'U';
        --Similar entries as above for all non output ports

        port_name : port_mode std_logic;
        --Similar entries as above for all output ports
    );
end component;
end SmartModels;
```

## Importing FMI Models into VHDL

You need to be aware of the following when you import FMI models into VHDL.

- You must use the `-fmient` option to specify the entity name of the foreign model.
- You must use the `-fmilib` option to specify the library in which the foreign model resides.
- You must specify the name of the foreign model.
- The component declaration is not needed for FMI models and it will not be generated.
- You must use the following command line syntax:

```
ncshell -import fmi -into vhdl -fmilib fmi_lib_name -fmient fmi_entity_name  
fmi_model_name
```

For example:

```
ncshell -import fmi -into vhdl -fmilib alu_lib -fmient alu_entity alu_model
```

The following shows the architecture shell that *ncshell* generates from the command line shown above.

### The FMP Model Architecture Shell

```
architecture fmi of fmi_entity_name is  
    attribute foreign of fmi:architecture is "FMI alu_lib :alu_model";  
begin  
end;
```

**Note:** If you use the `-backward` option to generate a shell that is compatible with Leapfrog, *ncshell* creates the following foreign attribute in the architecture:

```
attribute foreign of fmi : architecture is "alu_lib:alu_model";
```

## **ncsuffix**

The *ncsuffix* utility lets you display the machine architecture and the revision number of the library system for the current version of the software. You can display this information for the various intermediate objects that are generated by the simulator tools (*ncvlog*, *ncvhdl*, *ncelab*, *ncsim*).

You can use this information when you create Makefiles or otherwise automate your processes. For example, you can include a rule in a Makefile that checks to see if a library exists and if it is the right version. If the library is the wrong version, the rule can delete the old library.

### **ncsuffix Command Syntax**

Invoke *ncsuffix* with the `ncsuffix` command. One and only one option must be specified.

```
ncsuffix
  [-Ast]
  [-Cod]
  [-Help]
  [-NCError warning_code]
  [-NCFatal {warning_code | error_code}]
  [-NOcopyright]
  [-Pak]
  [-SIG]
  [-SSS]
  [-VErsion]
  [-VSt]
```

### **ncsuffix Command Options**

The following list shows the options you can use with the `ncsuffix` command. Options can be entered in upper or lower case. In the list, capital letters indicate the shortest possible abbreviation for an option.

#### **-Ast**

Prints the AST (Abstract Syntax Tree) suffix.

**-Cod**

Prints the COD suffix.

**-Help**

Prints a brief summary of the `ncsuffix` command-line options.

**-NCError *warning\_code***

Increase the severity level of the specified warning message from warning to error. The *warning\_code* argument is the message code (mnemonic) that appears in the warning message following the severity code.

You can include multiple `-ncerror` options on the command line.

Using this option can change the behavior of the tool because functions that return errors instead of warnings may behave differently.

**-NCFatal {*warning\_code* | *error\_code*}**

Increase the severity level of the specified warning message or error message from warning or error to fatal. The *warning\_code* or *error\_code* argument is the message code (mnemonic) that appears in the message following the severity code.

You can include multiple `-ncfatal` options on the command line.

**-NOcopyright**

Suppresses printing of the copyright banner.

**-Pak**

Prints the PAK file suffix.

**-Sig**

Prints the SIG suffix.

**-SSs**

Prints the *sss* (simulation snapshot) suffix.

**-VErsion**

Prints the version of *ncsuffix* and exits.

**-VSt**

Prints the *vst* (Verilog Syntax Tree) suffix.

## **Example ncsuffix Command Lines**

The following command prints the simulation snapshot suffix.

```
% ncsuffix -sss
ncsuffix: v2.2.(d5): (c) Copyright 1995 - 1999 Cadence Design Systems, Inc.
sun4v.090.sss
```

The following command suppresses printing of the copyright banner and prints the *vst* suffix.

```
% ncsuffix -nocopyright -vst
sun4v.090.vst
```

## **ncupdate**

When you change any of the design units in the hierarchy, you must compile and elaborate the design hierarchy again. You can automatically recompile and re-elaborate all out-of-date design units in the hierarchy with *ncupdate*. The *ncupdate* utility also calls the *ncsdfc* utility to recompile SDF source files if it detects that the SDF source has changed.

The argument to *ncupdate* is a snapshot name. Therefore, you must elaborate the entire design at least once before you can use *ncupdate*.

The purpose of *ncupdate* is to provide quick design change turnaround when you have edited a design unit. The modifications to design units cannot cross file boundaries to modify other files. Do not use *ncupdate* (or *ncsim -update*) after adding a design unit, a source file, or compiler directives to the design. For example, *ncupdate* will not update correctly if you edit a source file to define a new macro, or if you change a design unit in a way that introduces a new cross-file dependency. In these cases, recompile the design with *ncvlog -update*.

The *ncupdate* utility uses the same compiler and elaborator command line options that were used originally. If you want to use different command line options, recompile and re-elaborate by running *ncvlog* and *ncelab*.

You can also automatically recompile and re-elaborate all out-of-date design units by invoking the simulator with the *-update* option. See “[Updating Design Changes When You Invoke the Simulator](#)” on page 332.

### **ncupdate Command Syntax**

Invoke *ncupdate* with options and arguments. Options can occur in any order. Parameters to options must immediately follow the option they modify. Command-line options can be abbreviated to the shortest unique string, indicated here with capital letters.

```
ncupdate [options] [lib.]cell[:view]
  [-Append_log]
  [-Cdslib filename]
  [-ERRormax integer]
  [-EXCLFile filename]
  [-EXCLUde library_name]
  [-Force]
  [-HDLvar filename]
  [-HELP]
  [-Ieee]
  [-LIBrary library_name]
```

```
[-Loqfile logfile_name]
[-Messages]
[-NCError warning_code]
[-NCFatal {warning_code | error_code}]
[-NEverwarn]
[-NOCopyright]
[-NOLog]
[-NOREcompile]
[-NOSource]
[-NOSTdout]
[-NOWarn warning_code]
[-Overwrite]
[-SScript filename]
[-Show]
[-Unit module_name]
[-VERBOSE]
[-VERSion]
```

## **ncupdate Command Options**

The following list describes the options you can use with the *ncupdate* command. Options can be entered in upper or lower case. In the table, capital letters indicate the shortest possible abbreviation for an option.

### **-Append\_log**

Appends log information from multiple runs of *ncupdate* to one log file. This option is overridden by the *-nolog* option.

### **-Cdslib *filename***

Specifies the *cds.lib* file to use for this update.

### **-ERRormax *integer***

Aborts *ncupdate* after reaching the specified number of errors.

### **-EXCLFile *filename***

Excludes the libraries listed in the specified file when updating a model.

**-EXCLUde *library***

Excludes the specified library when updating a model. Excludes STD and IEEE libraries by default.

**-Force**

Forces an update whether or not the model is out of date.

**-HDIvar *filename***

Specifies the `hdl.var` file to use for this update.

**-HElp**

Prints a brief summary of the `ncupdate` command options. No other action besides printing the help message takes place.

**-IEEE**

Allows the IEEE library to be updated.

**-LLibrary *library***

Causes the compiler to use only the specified library when updating a model. All other libraries listed in your `cds.lib` file are excluded.

**-LOgfile *logfile\_name***

Specifies the name for the log file. This name is used instead of the default name `ncupdate.log`. You can not include this option in an `hdl.var` file. This option overrides the `-nolog` option.

**-Messages**

Prints informative messages during execution. Information is only written to the log file when you use the `-messages` option.

### **-NCError *warning\_code***

Increase the severity level of the specified warning message from warning to error. The *warning\_code* argument is the message code (mnemonic) that appears in the warning message following the severity code.

Example:

```
% ncupdate -messages my_lib.top:snap -ncerror ABCDEF
```

You can include multiple `-ncerror` options on the command line.

Using this option can change the behavior of the tool because functions that return errors instead of warnings may behave differently.

### **-NCFatal {*warning\_code* | *error\_code*}**

Increase the severity level of the specified warning message or error message from warning or error to fatal. The *warning\_code* or *error\_code* argument is the message code (mnemonic) that appears in the message following the severity code.

Example:

```
% ncupdate -messages my_lib.top:snap -ncfatal LMNOPQ
```

You can include multiple `-ncfatal` options on the command line.

### **-NEverwarn**

Disables printing of all *ncupdate* warning messages.

### **-NOCopyright**

Suppresses printing of the copyright banner.

### **-NOLog**

Do not generate a log file. By default, *ncupdate* generates a log file called `ncupdate.log`. This option is overridden by the `-logfile` option.

**-NOREcompile**

Used with `-show` to display the modules that need to be updated.

**-NOSource**

Eliminates source file time stamp checks.

**-NOSTdout**

Suppresses printing to the screen.

**-NOWarn *warning\_code***

Disables the printing of the specified *ncupdate* warning. The *warning\_code* is the sequence of characters following the error severity code.

**-Overwrite *script\_filename***

Causes the script file that is specified with the `-script` option to be overwritten. The `-overwrite` option is used with the `-script` option to replace a previously generated script.

**-SScript *filename***

Generates a script to recompile the model. Generates a script that forces an update of all units when used with the `-force` option.

**-Show**

Prints the compile commands to the screen. Use the `-show` option with the `-norecompile` option to display modules that need to be updated.

**-Unit *module\_name***

Updates the specified module. The default argument to *ncupdate* is a snapshot. Use the `-unit` option to recompile a design unit if the source code has changed.

**-VERBose**

Prints the reasons for a recompile.

**-VERSion**

Prints the version of *ncupdate* and exits.

**Example ncupdate Command Lines**

To update the module `my_lib.top:behav`

```
% ncupdate -unit my_lib.top:behav
```

To create a script to force update of all units in `my_lib`

```
% ncupdate -force -script ./update.script -library my_lib
```

To update the snapshot `my_lib.top:snap` with informative messages

```
% ncupdate -messages my_lib.top:snap
```

## shellgen

The *shellgen* utility generates a Verilog or VHDL model shell that you can instantiate in a design to import OMI-compliant models. The shell file is a Verilog module or a VHDL entity/architecture body pair that contains:

- A set of predefined attributes that identify the model and the corresponding model manager. The NC simulation tools use the attribute information to identify an imported OMI model, to locate the controlling model manager, and to initiate communication with it.
  - `foreign`—Identifies the Verilog module or VHDL entity/architecture as a shell for importing an OMI-compliant model.
  - `mm_path`—Specifies the path to the model manager object file.
  - `mm_object`—Specifies the name of the model manager object file.
  - `mm_bootstrap`—Specifies the name of the model manager bootstrap routine to call.
  - `model`—Specifies the name of the model.
  - `library`—Specifies the library name.

The OMI specification supports models delivered as either model objects or libraries. The `library` attribute is necessary when models with the same name are packaged into different libraries controlled by the same model manager.

- Port, parameter/generic, and signal declarations that *shellgen* has extracted from the model.
- Definitions of the *viewports* of the model, the internal signals (if any) that the model provider has designated as visible for read and/or write permission.

To generate the simulation shell, invoke *shellgen* with the `shellgen` command. You must use the `-b` option to specify the full path to a file supplied with the model manager called the *bootstrap* file. This file contains the name of the model manager object file and the name of the bootstrap routine to call.

When you invoke *shellgen*, it:

- Loads the appropriate model manager specified in the bootstrap file.
- Queries the model manager for the presence of the model(s) you specified.
- Extracts model boundary information (ports, parameters, generics, and viewports).

- Writes the shell file.

See “[The Open Model Interface \(OMI\)](#)” on page 952 for more information on importing OMI models. See [Chapter 21, “Cosimulation with NC-Verilog and Quickturn.”](#) for information on including models that simulate in conjunction with Quickturn Emulation Systems.

## **shellgen Command Syntax**

Invoke *shellgen* with options and arguments. Options can occur in any order. The syntax of the *shellgen* command is as follows:

```
shellgen [-b bootstrap_file] [other_options]  
[-f options_file]  
[-help]  
[-l library_name]  
[-m model_name]  
[-nomm_object]  
[-nomm_path]  
[-o output_file]  
[-pli]  
[-r]  
[-unresolved]  
[-verilog]  
[-version]  
[-vhdl]  
[+model_manager invocation option[=option_value]]  
[+quickturn mm option[=option_value]]
```

## **shellgen Command Options**

The following list describes the options that you can use with the *shellgen* command.

### **-b *path\_to\_bootstrap\_file***

Use the bootstrap file in the specified location. This option is required.

The bootstrap file is a text file supplied by the model provider. It contains:

- The name of the model manager object file.
- The name of the bootstrap routine to call so that *shellgen* can initiate communication with the model manager.

## NC-Verilog Simulator Help

### Utilities

---

The argument specifies the full path name to the bootstrap file. For example:

```
/net/machine/model_manager/bootstrap_file
```

#### **-f *options\_file***

Use the `shellgen` command options in the specified file.

The options file can contain only options that take a value. In the file, specify each option on a separate line. You must use the equal ( = ) sign in the options file (for example, `-l=library_name`).

#### **-help**

Print a brief summary of the `shellgen` command options.

#### **-l *library\_name***

Generate shells for all models in the specified library.

If you do not use the `-m model_name` option to specify a particular model, `shellgen` generates shells for all of the models in this library.

#### **-m *model\_name***

Generate a shell for the specified model.

The `model_name` argument is the name of the model as defined by the model provider. Model names are case sensitive.

If you omit this option, `shellgen` generates a shell for all models controlled by the model manager. Use the `-l library_name` option to specify that shells are to be generated for models in a particular library.

#### **-nomm\_object**

Generate the `mm_object` attribute value without a file extension in the shell file.

**-nomm\_path**

Generate an empty string for the `mm_path` attribute in the shell.

By default, the `mm_path` attribute value is the full path to the model manager object file. Using the `-nomm_path` option removes the dependency on a specific model manager install location.

If you use this option, you must include the path to the model manager dynamic library in your library path environment variable (`LD_LIBRARY_PATH` on Solaris or `SHLIB_PATH` on HP), so that the OMI socket can locate and load the library.

**-o *output\_file***

Generate a shell file with the specified filename.

If you do not use this option, shell files are created in the current working directory and are called:

- `model_name.v` for Verilog
- `model_name.vhd` for VHDL
- `model_name_pli.v` for use with the IP Model Packager OMI adapter.

If a file already exists, the shell is written to stdout unless you include the `-r` option.

**-pli**

Create a Verilog PLI shell for use with the IP Model Packager OMI adapter.

If you do not use this option or the `-vhdl` or `-verilog` option to specify the kind of HDL shell to create, `shellgen` creates a Verilog shell file for the model by default.

**-r**

Overwrite the existing shell file.

**-unresolved**

Use the `std_ulegic` and `std_ulegic_vector` unresolved types for ports and viewports of any logic type when creating a VHDL shell. The default is to use the `std_logic` and `std_logic_vector` logic types.

**-verilog**

Create a Verilog shell.

If you do not use this option or the `-vhdl` or `-pli` option to specify the kind of HDL shell to create, *shellgen* creates a Verilog shell file for the model by default.

**-version**

Display the version of *shellgen*.

**-vhdl**

Create a VHDL shell.

If you do not use this option or the `-verilog` or `-pli` option to specify the kind of HDL shell to create, *shellgen* creates a Verilog shell file for the model by default.

**+model\_manager\_invocation\_option[=option\_value]**

Use the specified model manager invocation option.

A model manager may define its own invocation options. The *shellgen* utility does not recognize these options, but can pass them to the model manager. See the model manager documentation for descriptions of any model manager invocation options.

Each model manager invocation option begins with a plus sign ( + ) followed by the name of the option. If the option takes a value, specify the value after the equal sign ( = ).

You can include model manager invocation options in an options file that you specify with the `-f` option. Each invocation option must be on a separate line.

### **+quickturn\_mm\_option [=option\_value]**

Use the specified Quickturn model manager invocation options.

Some model manager providers may require special invocation options for shell generation. For example, to include models that simulate in conjunction with the Quickturn Emulation System, two model manager options (+qt\_model and +qt\_pin\_map) are required for shell generation. See “[Cadence Model Manager for Quickturn Command-Line Plus Options](#)” on page 970 for a list of the model manager for Quickturn plus options.

You can include model manager invocation options in an options file that you specify with the -f option.

### **shellgen Command Example**

The following command line generates a shell file for including a model called testmodel for a simulation in conjunction with the Quickturn Emulation System. The -b option, which is required, specifies the full path to the model manager bootstrap file. The -o option is used to name the shell file shell\_ncv.v instead of testmodel.v. The plus options on the command line are Quickturn model manager invocation options.

```
% shellgen -b /QTinstall/tools/lib/qtbootstrap -o shell_ncv.v \
-nomm_path +qt_model=testmodel +qt_pin_map=top.map
+qt_mode=event +qt_emulator=qtr +qt_qbridge=QUICKTURN
```

The following shows the shell file, shell\_ncv.v, generated by this command:

```
module testmodel (memAddr, memDataIn, read, clk, memDataOut, nint, reset,
test_mode, scan_enable, scan_in, scan_out )
(* integer foreign = "OMI";
integer mm_path = " ";
integer mm_object = "qtmanager.so";
integer mm_bootstrap = "qt_manager";
integer model = "testmodel";
integer omi_unit_delay = 1;
*);
output [12:0] memAddr;
output [7:0] memDataIn;
output read;
input clk;
input [7:0] memDataOut;
input nint;
input reset;
```

## NC-Verilog Simulator Help

### Utilities

---

```
input test_mode;
input scan_enable;
input scan_in;
output scan_out;
parameter qt_mode = "event";
parameter qt_pin_map = "top.map";
parameter qt_emulator = "qtr";
parameter qt_qbridge = "QUICKTURN";
parameter qt_strobedelay = 15;
parameter qt_lib = "libqteapi";
reg (* integer omi_viewport = "READ"; *) \ul.foo;
endmodule
```

# The Programming Language Interface (PLI)

---

The Programming Language Interface (PLI) is a public-domain C-language procedural interface and interface mechanism that lets you access and modify data dynamically in the data structure that results from compiling Verilog HDL source descriptions and elaborating the design hierarchy. The PLI provides a library of C-language functions that can directly access data within the data structure.

Some applications of the PLI interface include:

- Customized debugging routines
- Applications that read data, such as test vectors, from a file and that pass the data to the simulator
- Simulation models written in C and dynamically linked into a Verilog simulation
- Event-driven callback routines
- Delay calculators
- Backannotating routines

The PLI has been standardized by the IEEE. The standard is described in the *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*.

The IEEE standard describes the PLI routines in terms of three generations:

- Task/function routines (TF routines)

These routines are used primarily for operations involving user-defined system task/function arguments and for utility functions, such as setting up callback mechanisms and writing data to output devices. The TF routines are sometimes called *utility* routines.

## NC-Verilog Simulator Help

### The Programming Language Interface (PLI)

---

#### ■ Access routines (ACC routines)

These routines provide access into a Verilog HDL structural description. The ACC routines are used to access and modify information, such as delay values and logic values on a wide variety of objects that exist in a Verilog description.

The TF and ACC routines are sometimes referred to as PLI 1.0.

In addition to the information on the PLI TF and ACC mechanism contained in the IEEE standard, refer to the [\*PLI 1.0 User Guide and Reference\*](#) for details on these routines.

#### ■ Verilog Procedural Interface (VPI) routines

These routines provide an object-oriented access for both Verilog HDL structural and behavioral objects. The VPI routines are a superset of the functionality of the TF and ACC routines.

Using the VPI rather than PLI 1.0 is strongly recommended because VPI is more complete, is easier to learn and to use, and can result in better simulation performance.

In addition to the information on the VPI contained in the IEEE standard, see the *VPI User Guide and Reference* for details on the VPI routines.

To use PLI with the NC-Verilog simulator, you write a standard C language application that calls PLI routines. You then integrate your application either by compiling and linking the application and the simulator object modules into a new set of executables (static linking), or by compiling and linking the application into a shared library (dynamic linking). If a PLI application has been compiled into a dynamic shared library, you can use the `-loadpli1` or `-loadvpi` command-line option to load the library and to register the system tasks defined in the application at run time.

There are two ways to integrate a PLI application with the NC-Verilog simulator:

- Use the PLI Wizard. This is a graphical interface that presents a series of windows that lead you through the process of linking the application. See the *PLI Wizard User Guide* for information on using the PLI Wizard.
- Manually copy and then edit the `Makefile.nc` file in your installation (UNIX) or create a `libpli.dll` (NT). See Chapter 3 in the *PLI Wizard User Guide* for details.

---

## Importing Foreign Models

---

The NC-Verilog simulator supports three modeling interfaces that let you integrate hardware models into your simulation: SmartModel SWIFT Interface, the LMSI Hardware Modeling Interface, and the OMI interface.

These interfaces are described in the following sections:

- [The SmartModel SWIFT Interface](#)
- [The Hardware Modeling Interface \(LMSI\)](#)
- [The Open Model Interface \(OMI\)](#)

## The SmartModel SWIFT Interface

The SmartModel SWIFT™ Interface lets you create and simulate designs that include SmartModel Library models provided by the Logic Modeling Group (LMG) of Synopsys. Using the interface, you can instantiate SmartModel Library models in your Verilog HDL design and monitor and modify their contents.

The SWIFT Interface is available on UNIX and Windows NT platforms.

The NC-Verilog simulator supports SmartModels R41.

For more details on SmartModel Library models, see the Synopsys *SmartModel Library Reference Manual*.

## Using the SmartModel SWIFT Interface with NC-Verilog

LMG provides two description files for each model:

- An object file (`modelname.so` on UNIX, or `modelname.dll` on NT) that contains the object code for the model
- A Verilog HDL shell file (`modelname.v`) that contains a Verilog HDL description of the model

To use a SmartModel Library model in a Verilog HDL design that you simulate with the NC-Verilog simulator, you:

- Install the library models and the SWIFT interface.
- Integrate the SWIFT Interface with NC-Verilog.
- Instantiate library models in your design.
- Compile the source files, including the `.v` files for the library models.
- Elaborate and simulate the design.

## Integrating SmartModel Library Models with NC-Verilog on UNIX

There are two ways to integrate the SWIFT interface with the NC-Verilog simulator:

- Use the PLI Wizard. This is a graphical interface that presents a series of windows that lead you through the process of linking the interface. See the *PLI Wizard User Guide* for information on using the PLI Wizard.

## NC-Verilog Simulator Help

### Importing Foreign Models

---

- Manually copy and then edit the `Makefile.nc` file in your installation (UNIX) or create a `libpli.dll` (NT). The rest of this section tells you how to do this.

On UNIX platforms, you integrate the SWIFT interface with the NC-Verilog simulator by statically linking the following components with the elaborator and with the simulator to create a new set of NC-Verilog executables:

- Available in the Cadence installation:
  - The NC-Verilog object modules: `ncelab.o`, `ncsim.o`
- Available in the LMC installation:
  - The LMC shared library (`lmtv.o`)
  - The SWIFT interface (`sun4Solaris.lib` or `hp700.lib`)

To build the new NC-Verilog executables:

1. Copy the file `Makefile.nc` to a `Makefile` in your application directory. Call the file `Makefile` (not `Makefile.nc`).

`Makefile.nc` is available in the Cadence installation in the directory:

`your_install_directory/tools/inca/files`

2. Edit `Makefile`.

- Edit the `LMC_HOME` macro to point to the SWIFT installation.
- If you are using a version prior to R41, edit the `LMC_LIB` macro, changing `lmtv.o` to `lmtv.a`.

**Note:** The shared VPI and PLI libraries are linked in because they contain global variables referenced by the NC-Verilog simulator. If you are building a PLI or VPI application in addition to using a SmartModel Library model, you must adjust the `Makefile` as described in the [PLI 1.0 User Guide and Reference](#) or [VPI User Guide and Reference](#) manuals, respectively.

3. Set the `LD_LIBRARY_PATH` (Solaris platform) or the `SHLIB_PATH` (HP platform) environment variable as follows:

```
setenv LD_LIBRARY_PATH  
your_install_directory/tools/inca/lib:/usr/dt/lib:usr/lib  
setenv SHLIB_PATH your_install_directory/tools/inca/lib:usr/lib
```

4. Type `make swift` to execute the branch of the `Makefile` that builds `ncelab` and `ncsim`.

## Integrating SmartModel Library Models with NC-Verilog on Windows NT

On Windows NT, the SWIFT interface is linked in as a PLI application. The LMG software includes a dynamic link library, lmtv.dll with an import library lmtv.lib, which has been linked against the Cadence pliinterface.lib.

After installing the software from LMG, all you have to do is to create a libpli.dll that links in lmtv.lib and modify your PATH variable so that it includes the path to libpli.dll.

To create the required libpli.dll, follow these steps:

1. Copy the veriuser.c file from the *your\_install\_dir/tools/src* directory into your local working area.
2. Edit the copy of veriuser.c as follows:

- a. After #include vxl\_veriuser.h, add the following line:

```
#include ccl_lmtv_include.h
```

- b. After /\*\*\*\* add user entries here \*\*\*\*/, add the following line:

```
#include ccl_lmtv_include_code.h
```

- c. Add any of your own code needed to support PLI code that you want to add.

3. Use the following command to compile veriuser.c to create veriuser.obj:

```
cl -O2 -MD -DMSC -DWIN32 -Iyour_install_dir/tools/include -I%LMC_HOME%/include  
-c veriuser.c -Foveriuser.obj
```

**Note:** In the -O2 expression at the beginning of the command line, the first letter is a capital O, not a zero.

4. Create libpli.dll as follows:

```
link -DLL -out:libpli.dll veriuser.obj %LMC_HOME%/lib/pcnt.lib/lmtv your_libs  
your_obj
```

You may need to move the libpli.dll if it is not in one of your NT search paths.

%LMC\_HOME%/lib/pcnt.lib must be in your PATH variable so that libpli.dll can find and load lmtv.dll.

## Running SmartModel Library Models with NC-Verilog

To run the NC-Verilog simulator on a design that includes SmartModels library models, compile your source files, including the `.v` file(s) of the model(s) you used in your design, elaborate the design with `ncelab`, and run `ncsim`. For example, assume that you have a top-level module called `top` in a file called `my_hdl.v`, and that module `top` includes a model called `TTL260`. Run the NC-Verilog simulator as follows:

### 1. Run the `ncvlog` compiler:

```
% ncvlog my_hdl.v TTL260.v
```

The compiler takes as input your Verilog HDL design and the SmartModel library model HDL description, and performs syntactic and semantic checking. It generates an intermediate representation for the design.

**Note:** If you use two or more SmartModel Library models in your design, you can list them in a `.v` file (`swift_mods.v`, for example). When you run `ncvlog` to compile the source files, you can include this file using the `-f` option as follows:

```
% ncvlog my_hdl.v -f swift_mods.v
```

### 2. Run the `ncelab` elaborator:

```
% ncelab top
```

The elaborator takes as input the name of the top-level design unit (in this example `top`) and constructs a design hierarchy based on the instantiation and configuration information in the design.

### 3. Run the `ncsim` simulator:

```
% ncsim top
```

## The Hardware Modeling Interface (LMSI)

Cadence's Logic Modeling Group (LMG) Hardware Modeling Interface lets you create and simulate designs that include LMG (Logic Modeling Group) hardware models. The interface is the layer of communication between the simulator and the LMG module SFI, which, in turn, communicates with the hardware model(s).

The Hardware Modeling Interface is a separate software module from the simulator. You integrate the interface and the SFI module with the elaborator and with the simulator to create a new set of NC-Verilog executables that can communicate with the hardware models.

### Using LMG Hardware Models with NC-Verilog

The hardware models are physically located in a hardware modeler device. LMG provides:

- A `.mdl` file that contains a hardware description of the model.
- The Simulator Function Interface (SFI) module, which enables the communication between the LMG Hardware Modeling Interface and the hardware modeler.

The interface enables the communication between the simulator and the SFI module which, in turn, communicates with the hardware modeler.

Cadence provides a utility called `crsshell` that takes as input the `.mdl` model description and automatically generates a Verilog HDL model description, which you run with the NC-Verilog simulator.

To use an LMG hardware model in a Verilog HDL design that you simulate with the NC-Verilog simulator, you:

- Integrate the LMG Hardware Modeling Interface and the SFI library with NC-Verilog.
- Use the Verilog HDL description of the model, generated using `crsshell`, to run it with NC-Verilog.

For more details on the LMG Hardware Modeling Interface and the `crsshell` utility, see the Cadence *Hardware Modeling Interface Reference Manual and User Guide*.

For more details on LMG hardware models and the SFI module, see the Synopsys *LM-Family Modeler* documentation.

## Integrating the LMG Hardware Modeling Interface with NC-Verilog

There are two ways to integrate the LMG Hardware Modeling interface with the NC-Verilog simulator:

- Use the PLI Wizard. This is a graphical interface that presents a series of windows that lead you through the process of linking the interface. See the *PLI Wizard User Guide* for information on using the PLI Wizard.
- Manually copy and then edit the `Makefile.nc` file in your installation. The remainder of this section tells you how to do this.

You integrate the LMG Hardware Modeling Interface with the NC-Verilog simulator by statically linking the following components into a new set of NC-Verilog executables:

- Available in the Cadence installation
  - The NC-Verilog object modules: `ncelab.o`, `ncsim.o`
  - The LMG Hardware Modeling Interface object module: `nclm_vlog.o`
- Available in the LMG installation
  - The SFI static library module: `lm_sfi.a`

You take the following steps to build the new NC-Verilog executables:

1. Copy the file `Makefile.nc` to a Makefile in your application directory.

`Makefile.nc` is available in the Cadence installation in the directory:

`your_install_directory/tools/inca/files`

2. Edit your copy of `Makefile.nc`. Edit the `SFI_ROOT` macro to point to the LMG static library installation directory.

**Note:** The default `veriuser.o` and `vpi_user.o` object files are built and linked in because they contain global variables referenced by the NC-Verilog simulator. If you are building a PLI or VPI application in addition to using an LMG hardware model, you must edit the `veriuser.c` or `vpi_user.c` files as described in the [\*PLI 1.0 User Guide and Reference\*](#) or the [\*VPI User Guide and Reference\*](#) manuals, respectively.

3. Set the `LD_LIBRARY_PATH` (Solaris platform) or the `SHLIB_PATH` (HP platform) environment variable as follows:

```
setenv LD_LIBRARY_PATH  
your_install_directory/tools/inca/lib:/usr/dt/lib:usr/lib  
setenv SHLIB_PATH your_install_directory/tools/inca/lib:usr/lib
```

4. Type `make lmsi` to execute the rules in `Makefile.nc` that build the new `ncelab` and `ncsim` executables

## Running LMG Hardware Models with NC-Verilog

You run the NC-Verilog simulator with your main Verilog HDL design file and the `.v` file of the hardware model you used in your design, which you have generated using the `crshell` utility.

For example, for a main design in a file called `my_hdl.v`, with a top module called `top`, that includes the model `TIALS08`, you would run the NC-Verilog simulator as follows:

1. Run the `ncvlog` compiler:

```
% ncvlog my_hdl.v TIALS08.v
```

The compiler takes as input your Verilog HDL design and the HDL description of the LMG hardware model and performs syntactic and semantic checking. It generates an intermediate representation for the design.

**Note:** If you are using two or more LMG hardware models in your design, you can list them in a `.v` file (for example, `lmsi_mods.v`). When you run `ncvlog` to compile the source files, you can include this file using the `-f` option as follows:

```
% ncvlog my_hdl.v -f lmsi_mods.v
```

2. Run the `ncelab` elaborator:

```
% ncelab top
```

The elaborator takes as input the name of the top-level design unit (in this example `top`) and constructs a design hierarchy based on the instantiation and configuration information in the design.

3. Run the `ncsim` simulator:

```
% ncsim top
```

## Specifying the Delay Mode for LMG Hardware Models

To specify the delay mode for the LMG hardware models in your design, enter it as a `+argument` on the simulator command line. So, to specify delays on the command line for Verilog HDL designs that include LMG hardware models, follow these steps:

1. Specify the delay on the `ncelab` command line

```
% ncelab -mindelays top
```

2. Specify the delay again on the `ncsim` command line

```
% ncsim +mindelays top
```

## NC-Verilog Simulator Help

### Importing Foreign Models

---

The delay mode must be specified again in the simulator because the LMG hardware models scan the + arguments passed to the simulator to set their delay mode.

**Note:** If no delay is specified when you invoke the simulator, the simulator will run using the LMG model default delay, `maxdelays`. If no delays are specified in either the elaborator or simulator steps, the elaborator will run using the HDL design default delay, `typdelays`, and the simulator will run using the LMG model default delay, `maxdelays`.

## The Open Model Interface (OMI)

The Open Model Interface (OMI) is an open procedural interface that lets you use models that have been compiled and packaged using an OMI-compliant packaging tool with an OMI-compliant simulator, such as the NC-Verilog simulator or the NC-VHDL simulator, regardless of the language in which the models have been developed.

The OMI model import capability in the NC simulators lets you import OMI models that are compliant with Version 4.0 of the IEEE Std. 1499 Open Model Interface specification. Using this capability you can:

- Reuse OMI-based simulation models in your design. Using an OMI-compliant model packaging solution, IC vendors can deliver models developed in an HDL modeling language, such as Verilog or VHDL, or in a general-purpose programming language, such as C. You can then use the packaged models in your design. Models are delivered in object code format to protect intellectual property.
- Include models that simulate in conjunction with the Quickturn emulation system. See [Chapter 21, “Cosimulation with NC-Verilog and Quickturn,”](#) for details on Quickturn.
- Include models that were packaged with the IP Model Packager. These are high-performance protected binary representations of HDL models.
- Reuse the same stimulus/testbench developed in SPW that was used by high-level system designers to model the environment surrounding the target design. You can import these system testbenches and use them to help you verify that the hardware you are developing in an HDL is functionally correct.
- Reuse models that were developed by system designers to describe the algorithms or specification used in the target design. To help you verify that your HDL design matches the original system design algorithm/specification, you can import the original models into the design and use them as an embedded reference (shadow) component.

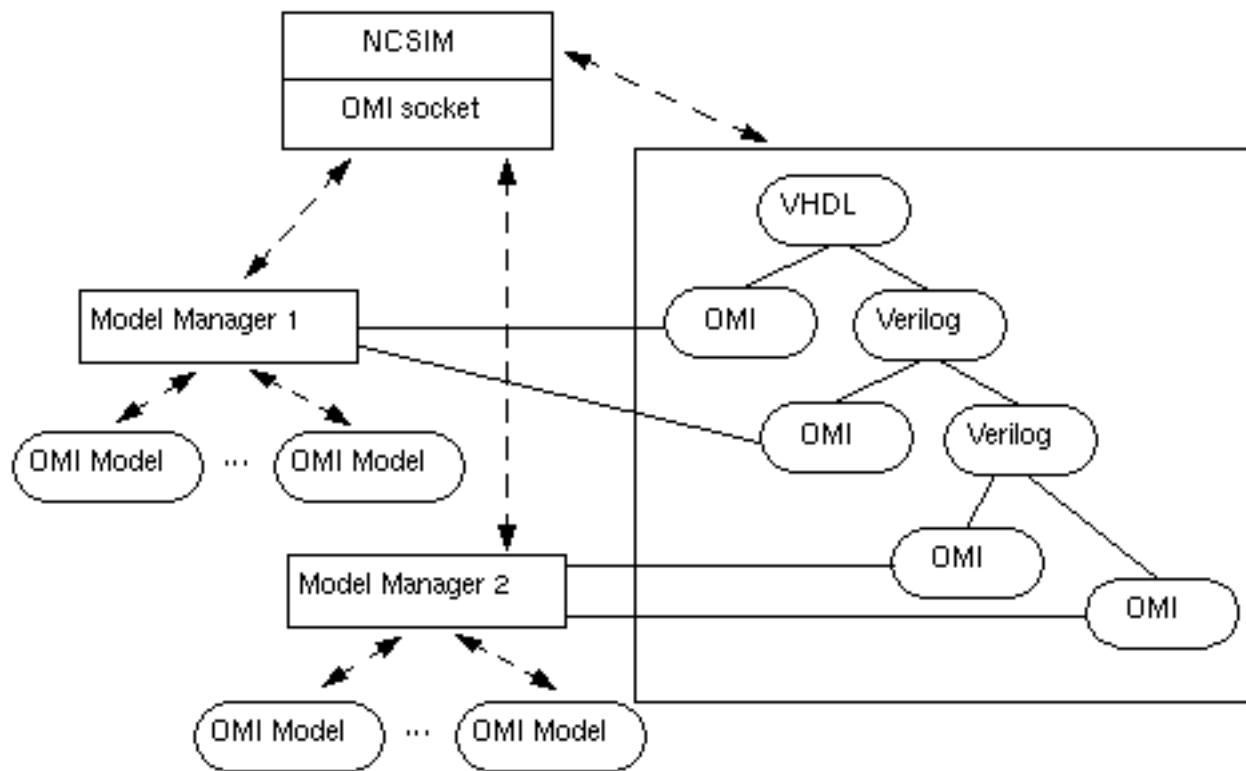
OMI models are integrated into a design as black-boxed models through an HDL model shell. The model shell defines the ports and design parameters of the model without defining the structure or behavior of the model. The shell may also contain definitions of *viewports*, the internal signals that the model provider has designated as visible for read and/or write permission. These internal signals provide limited visibility into the model.

The OMI interface communicates boundary information and activities with model instances through a *model manager*. Each OMI model is controlled by a model manager. A model manager can control multiple models, and the simulator can communicate with more than

## NC-Verilog Simulator Help

### Importing Foreign Models

one model manager in a single simulation session. The following figure shows the major software elements in a mixed-language simulation that includes imported OMI models:



## Integrating OMI Models

To integrate an OMI model into a design, you have to:

1. Install the model(s) and the model manager. See the documentation provided by the model supplier for details on the installation procedure.
2. Use the *shellgen* utility to create a Verilog or VHDL model shell to represent the model in the design.

The shell file is a Verilog module or a VHDL entity/architecture pair that serves as a wrapper for integrating the OMI model into the target Verilog or VHDL design as a foreign model.

**Note:** The IP Model Packager runs *shellgen* to create shells for the models being packaged.

3. Instantiate the Verilog module or the VHDL entity/architecture pair contained in the shell in the Verilog or VHDL design.

4. Compile the shell and the design source files, elaborate the design, and then simulate the design.

**Note:** In order to simulate OMI models that are controlled by a C++ model manager, you must create a new elaborator and simulator that are linked with a C++ compiler rather than with a C compiler. See “[Simulating OMI Models Controlled by C++ Model Managers](#)” on page 966 for details.

The following section provides general information about shells that is applicable to both Verilog and VHDL. See “[Integrating an OMI Model into a Verilog Design](#)” on page 955 for details on integrating an OMI model into a Verilog design and “[Integrating an OMI Model into a VHDL Design](#)” on page 959 for details on integrating an OMI model into a VHDL design.

## Generating a Model Shell

You integrate OMI models into a design using an HDL model shell.

To generate a shell, use the *shellgen* utility. You can generate a shell for one model, or you can generate shells for all models in a specified library of models. See “[shellgen](#)” on page 934 for details on running *shellgen*.

A model shell contains:

- A set of predefined attributes that identify the model and the corresponding model manager. The NC simulation tools use the attribute information to identify an imported OMI model, to locate the controlling model manager, and to initiate communication with it. The predefined attributes are:
  - ❑ `foreign`—Identifies the Verilog module or VHDL entity/architecture as a shell for importing an OMI-compliant model.
  - ❑ `mm_path`—Specifies the path to the model manager object file.

You can use the `-nomm_path` option when you run *shellgen* to generate an empty string for the `mm_path` attribute in the shell. Using this option removes the dependency on a specific model manager install location. If you use this option, you must include the path to the model manager shared object in your library path environment variable (`LD_LIBRARY_PATH` on Solaris or `SHLIB_PATH` on HP), so that the OMI socket can locate and load the library.

- ❑ `mm_object`—Specifies the name of the model manager object file.
- ❑ `mm_bootstrap`—Specifies the name of the model manager bootstrap routine to call.
- ❑ `model`—Specifies the name of the model.

- library—Specifies the library name.

The OMI specification supports models delivered as either binary objects or as sharable libraries.

Several models can be grouped together as a library. The `library` attribute is necessary when models with the same name are packaged into different libraries controlled by the same model manager.

- Definitions of the ports and design parameters of the model.
- Definitions of the *viewports*. Viewports are the internal signals (if any) that the model provider has designated as visible for read and/or write permission. Refer to the user documentation supplied by the model provider for details on viewport access.

Different restrictions apply to viewport use in VHDL and Verilog designs. These restrictions are covered in the following sections.

The shell may also include descriptive information in the form of comments to help you use the model correctly. For example, comments may tell you how to interpret the vector bits.

## Integrating an OMI Model into a Verilog Design

A Verilog module shell that defines the model ports and parameters is used to instantiate an OMI model in a Verilog design. The following example shows a Verilog shell for an OMI model generated by the IP Model Packager. This example shell has been edited to save space.

```
module rpuTop (A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,
A14,A15,iom,rd,sdo,wr,Dout0,Dout1,Dout2,Dout3,Dout4,Dout5,Dout6,
Dout7,Dout8,Dout9,Dout10,Dout11,Dout12,Dout13,Dout14,Dout15,clock,
reset,sdi,se,stall)
(* integer foreign      = "omi";
   integer mm_path      = "";
   integer mm_object     = "libcdsimm1.0.d4.so";
   integer mm_bootstrap  = "mmAffirmaBootstrap";
   integer model         = "rpuTop";
*)
output A0 ;      // omimVVL4
output A1 ;      // omimVVL4
...
...
output sdo ;    // omimVVL4
output wr ;     // omimVVL4
inout Dout0 ;   // omimVVL4
inout Dout1 ;   // omimVVL4
```

```
...
...
inout Dout15 ; // omiMVL4
input clock ; // omiMVL4
input reset ; // omiMVL4
input sdi ; // omiMVL4
input se ; // omiMVL4
input stall ; // omiMVL4
reg [15:0] (* integer omi_viewport = "READ"; *) Din ;
reg [15:0] (* integer omi_viewport = "READ"; *) Dout ;
reg [15:0] (* integer omi_viewport = "READ"; *) PC ;
reg (* integer omi_viewport = "READ"; *) writeOk ;
endmodule
```

Logic viewport objects are represented as registers in the Verilog module shell. Viewports of omi1164Logic and omi1364Logic data types are converted to MVL4 (0,1,Z,X) based on the VHDL and Verilog logic value mappings defined in the OMI specification.

**Note:** Time and memory viewports are not supported in the current release.

Verilog viewport access is specified in the model shell using the predefined `omi_viewport` attribute. The attribute value is set to "READ" for a read only viewport, and is set to "WRITE" for a viewport that has read and write access. For example:

```
reg [3:0] (* integer omi_viewport = "READ"; *) probe2;
reg (* integer omi_viewport = "WRITE"; *) \u1.inA ;
```

All model boundary objects (ports and viewports) have read access by default, even if the rest of the design has read/write/connectivity access turned off.

You can display and monitor viewport values by using simulator commands, such as `value` and `probe` (*Show—Value* and *Set—Probe* on the SimControl window), or by using Verilog system tasks such as `$display` and `$monitor`.

Use the `deposit` and `force` simulator commands (*Set—Deposit* and *Set—Force* on the SimControl window) to write to viewports. Viewport assignment is not allowed in an HDL design source file regardless of the access mode specified for the module shell.

OMI ports may be double precision floating point ports. These ports are typically marked with the following comment in the model shell:

```
// omiReal
```

For example:

```
module omitest (STROBE_,omitest_out,omitest_in)
(* integer foreign      = "omi";
   integer mm_path      = "./ModelManager/";
   integer mm_object     = "libspwMM_c40.so";
   integer mm_bootstrap  = "bootstrap_c40";
   integer model         = "omitest";
*)
parameter gain = 1.000000;
input STROBE_ ;           // omiMVL2
output [0:63] omitest_out ; // 64-bit real    // omiReal
input [0:63] omitest_in ; // 64-bit real    // omiReal
endmodule
```

Although the OMI socket automatically converts a logic vector to a floating point number when communicating with an OMI model with floating point ports, you must make sure that the bits represent a valid floating point number. A common error is to forget to initialize the wire to a non-X value.

In order to manipulate floating point ports as floating point numbers rather than as 64-bit vectors, you can use the `$bitstoreal` and `$realtobits` system tasks to convert the vector representation to a floating point representation or to convert a floating point representation to a vector representation. This conversion is particularly useful in `$display` and `$monitor` system tasks.

The following two examples illustrate floating point port connection. Each connects to the model shell with floating point points shown above.

### **Example 1:**

This example maintains corresponding real registers for simple floating point manipulation in behavioral code.

```
`timescale 1 ns / 1 ns
module omitest_tb;
reg [1:1] STROBE;
wire [0:63] spw_model_in;
wire [0:63] spw_model_out;
omitest spw_model(.omitest_in(spw_model_in), .omitest_out(spw_model_out),
                  .STROBE_(STROBE));
defparam spw_model.gain = 1.5;
real rspw_model_out,
      rspw_model_in;
assign spw_model_in = $realtobits(rspw_model_in);
```

## NC-Verilog Simulator Help

### Importing Foreign Models

---

```
always @(spw_model_out)
    rspw_model_out = $bitstoreal(spw_model_out);
initial
begin
    STROBE <= 0;
    $display("           Time      in      out");
    $monitor ($time,"%f %f",rspw_model_in,rspw_model_out);
    # 10 rpw_model_in <= 1.0;
    # 20 rpw_model_in <= 2.0;
    # 70 $finish; // stop at 100 ns
end
always
# 10 STROBE = !STROBE;
endmodule
```

#### **Example 2:**

This example uses `$bitstoreal` and `$realtobits` directly in the behavioral code. In this case, an explicit initialization of the wire connected to the input port is required.

```
`timescale 1 ns / 1 ns
module omitest_tb;
reg [1:1] STROBE;
reg [0:63] spw_model_in;
wire [0:63] spw_model_out;
omitest spw_model(.omitest_in(spw_model_in), .omitest_out(spw_model_out),
                 .STROBE_(STROBE));
defparam spw_model.gain = 1.5;
initial
begin
    STROBE <= 0;
    spw_model_in <= $realtobits(0.0);
    $display("           Time      in      out");
    $monitor ($time,"%f %f",
              $bitstoreal(spw_model_in),$(bitstoreal(spw_model_out)));
    # 10 spw_model_in <= $realtobits(1.0);
    # 20 spw_model_in <= $realtobits(2.0);
    # 70 $finish; // stop at 100 ns
end
```

```
always
begin
# 10 STROBE = !STROBE;
end
endmodule
```

By default, unit delay port updates scheduled by the model manager are treated as zero delay updates scheduled to occur in the next delta cycle. You can specify an actual delay for a model by adding an `omi_unit_delay` attribute to the Verilog module shell to indicate the time of the delay. The specified delay unit is based on the simulation timescale that applies to the model instance. In the following example, the specified unit delay is 1 ns.

```
'timescale 1ns/1ns
module omi_model (...)
(* integer foreign      = "OMI";      // <model manager name>
integer mm_path        = "/net/machine/spw_mm/";
integer mm_object       = "libspwMM_1.0";
integer mm_bootstrap    = "mmSPWB_1_0";
integer model           = "shiftfxp";
integer omi_unit_delay = 1;
*) ;
...
...
endmodule
```

## Integrating an OMI Model into a VHDL Design

A VHDL shell that contains an entity declaration/architecture body pair with the appropriate name, generics, ports, and signal declarations is used for encapsulating an OMI model in a VHDL design.

**Note:** While the version of `shellgen` shipped with the current release of the simulator generates shells that can also be used by the Leapfrog VHDL simulator, older Leapfrog shells will not work with the NC-VHDL simulator.

The following example shows a VHDL shell for a model generated by the IP Model Packager. This example shell has been edited to save space.

## NC-Verilog Simulator Help

### Importing Foreign Models

---

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY rpuTop IS
    PORT (
        A0 : OUT std_logic;
        A1 : OUT std_logic;
        ...
        ...
        A15 : OUT std_logic;
        iom : OUT std_logic;
        rd : OUT std_logic;
        sdo : OUT std_logic;
        wr : OUT std_logic;
        Dout0 : INOUT std_logic;
        ...
        ...
        Dout15 : INOUT std_logic;
        clock : IN std_logic;
        reset : IN std_logic;
        sdi : IN std_logic;
        se : IN std_logic;
        stall : IN std_logic
    );
END;
ARCHITECTURE omi OF rpuTop IS
    ATTRIBUTE foreign OF omi : ARCHITECTURE IS "OMI:rpuTop";
    --- libcdsmmm1.0.d4.so
    ATTRIBUTE mm_path : string;
    ATTRIBUTE model : string;
    ATTRIBUTE model OF omi : ARCHITECTURE IS "rpuTop";
    ATTRIBUTE mm_bootstrap : string;
    ATTRIBUTE mm_object : string;
    ATTRIBUTE mm_path OF omi : ARCHITECTURE IS "";
    ATTRIBUTE mm_object OF omi : ARCHITECTURE IS "libcdsmmm1.0.d4.so";
    ATTRIBUTE mm_bootstrap OF omi : ARCHITECTURE IS "mmAffirmaBootstrap";
BEGIN
    viewports : PROCESS
        VARIABLE Din :std_logic_vector(15 DOWNTO 0); -- readonly
        VARIABLE Dout :std_logic_vector(15 DOWNTO 0); -- readonly
        VARIABLE PC :std_logic_vector(15 DOWNTO 0); -- readonly
```

## NC-Verilog Simulator Help

### Importing Foreign Models

---

```
VARIABLE writeOk :std_logic; -- readonly
BEGIN
WAIT;
END PROCESS viewports;
END;
```

OMI parameters are represented as generics in the VHDL shell.

Viewport access is specified through a separate `viewports` process created in the architecture of the model shell. All the viewports of the OMI model are mapped to variables within this process. The process has no significance other than declaring the viewport variables.

The mapping of OMI viewport types to VHDL is shown in the following table:

OMI Viewport Type	VHDL Variable Type
omi1164LogicData	Std_logic/Std_ulogic
omi1364LogicData	
omiMVL4LogicData	
omiFixedArrayData	Std_logic_vector
omiMVL2LogicData	Bit
omiBooleanData	Boolean
omiIntegerData	Integer
omiRealData	Real
omiTimeData	Time

For all viewports of type `omiMemoryKind`, the `viewports` process declarative region in the shell contains a type declaration, `memory_memCount`, where `memCount` signifies every viewport of type `omiMemoryKind`.

You can display and monitor viewport values by using simulator commands, such as `value` and `probe` (*Show—Value* and *Set—Probe* on the SimControl window). Unlike Verilog, VHDL does not support a `$monitor` system task to monitor viewport values. To continuously monitor and display any change in probed objects, use the `probe -screen` simulator command.

Use the `deposit` and `force` simulator commands (*Set—Deposit* and *Set—Force* on the SimControl window) to write to viewports. Viewport assignment is not allowed in an HDL design source file.

By default, unit delay port updates scheduled by the model manager are treated as zero delay updates scheduled to occur in the next delta cycle. To specify an actual delay for a model, add the time-valued attribute `omi_unit_delay` declaration and specification in the VHDL foreign architecture. In the following example, the specified unit delay is 1 fs.

```
architecture omi of <model> is
    attribute foreign ...
    ...
    attribute omi_unit_delay : time;
    attribute omi_unit_delay of omi:architecture is 1 fs;
end;
```

## Modifying a Model Shell

The interface of an exported model is defined by the model provider and is fixed. Do not modify the model shell to change or rearrange the port definitions.

The position, name, and value type of a Verilog parameter or VHDL generic object are fixed and must not be changed.

Default values are always generated for the parameters or generic objects in the shell. In some cases, you may have to edit the default value because an appropriate default value cannot be delivered with the model. For example, a parameter representing the pathname to a file on the local file system might require a default value that is peculiar to your site. In these cases, documentation explaining the meaning of the parameter and what values are appropriate should be provided by the model supplier.

If you change the default value of a Verilog parameter or VHDL generic, the value expression cannot not be altered in a way that affects the OMI parameter or generic type to which the original default value was mapped. Although Verilog does not recognize “types”, the default value expression in the shell must be consistent in representation with the default value specified by the model provider. For example, if the original expression is a string, the new default expression must also be a string. From an OMI perspective, a Verilog default expression can be classified as a string, an integer, or a real number.

## Simulating a Design With Imported OMI Models

After generating the shell and instantiating it in your design, compile your source files, including the shell, and then elaborate the design as usual. No special command-line options are required to simulate designs with imported OMI models.

### Scanning of Model Manager Invocation Options on the `ncsim` Command Line

You can specify model manager invocation options when you invoke the simulator with the `ncsim` command. The simulator does not interpret these options; it merely provides a mechanism to pass the invocation options to the model manager.

Specify model manager invocation options as plus options. The syntax is:

```
% ncsim +model_manager_option_keyword[=option_value] ...
```

The `model_manager_option_keyword` is defined by the model manager and is followed by a white space. The keyword matching operation is case-sensitive.

You provide the `option_value` (if required) as a string terminated by white space. The value is passed to the model manager without modification. Use quotation marks if the `option_value` contains white spaces. For example:

```
% ncsim +mmA_simopts="-input control.txt -log /tmp/mm.log"
```

### The `-omicheckinglevel` Option

Use the `-omicheckinglevel` option to specify the level of OMI checking to perform when you invoke the elaborator or simulator. The argument to this option can be:

- `max`—maximum checking level. Use this level for early integration testing and to debug problems, and then lower the level to increase performance.
- `std`—standard checking level. This is the default.
- `min`—minimum checking level. Select this level when you simulate to achieve higher performance after problems have been debugged. This level is not recommended during elaboration.

Examples:

```
% ncelab -omicheckinglevel max top
% ncsim -omicheckinglevel min worklib.top
```

## Simulator Commands

Imported OMI models are essentially black boxes. Some models may provide limited internal visibility through viewports. All ports and viewports have read access by default, even if the rest of the design has read, write, and connectivity access turned off.

You can access port and viewport values using simulator commands, such as `value` and `probe` (*Show—Value* and *Set—Probe* from the SimControl window), or with Verilog system tasks, such as `$display` and `$monitor`. For VHDL, the `probe -screen` command lets you monitor signals as they change value. These commands and system tasks show the effective values of ports and viewports. Values driven by the OMI model instance can only be accessed using the `drivers` command.

Some viewports may also have write access. Use the `deposit` or `force` command to assign and force values to ports and viewports. Viewport assignment is not allowed in an HDL design source file regardless of the access mode specified for the module shell.

Most simulator commands can be executed on OMI model objects. However, some simulator commands are affected by OMI model import.

- The `save` and `restart` commands are not supported in the current release. You cannot save or restart a simulation if it includes OMI models.  
A save or restart request results in a model manager warning message.
- You can use the `deposit` and `force` commands to assign or force values to OMI model ports and viewports only if the model provider has given write access to those objects. The model shell indicates if a viewport has read or write access.
- You can only use the `reset` command, which resets the currently loaded snapshot to its original state at time zero, if all model managers loaded for the current simulation session also support the reset feature.
- Inout and output ports declared within the OMI module shell are driven externally by the OMI model instances. A `scope -drivers` command on one of these objects shows the driver as a port. No driver information is available for viewports.

## The `omi` Command

Some model managers provide special capabilities that enhance the usability of the models under their control. For example, a model manager may let you load the contents of a memory viewport from a file, or it may let you dynamically control the collection of simulation data. The `omi` simulator command lets you pass model manager run-time commands to model managers that support this capability.

The `omi` command has two modifiers:

■ `-list`

Use this modifier to display information about the model managers and model instances for the current simulation session.

■ `-send`

Use this modifier to send commands to model managers and model instances. The simulator sends the specified command string to the model manager without modification.

See “[omi](#)” on page 568 for details on the `omi` command. See the documentation from the model manager provider for information on the model manager commands.

### **The `$omiCommand` System Task**

You can use the `$omiCommand` system task in your Verilog code to pass model manager commands to OMI instances. The syntax for this task is:

```
$omiCommand(instance_name, "command");
```

The `instance_name` must be an OMI instance. The command can be any command string supported by the model instance.

For example, the Cadence Model Manager for Quickturn lets you load the contents of a memory instance from a file. You can do this using a `$omiCommand` task, such as the following:

```
$omiCommand(testFix.DUT, "writemem ui.mem file.mem 16");
```

### **Message Logging**

The simulator displays a list of all model managers included for the simulation session before the simulation starts. Each model manager is listed with its name and an alias. The model manager aliases are used for sending commands to the model managers.

A model manager may create its own log file to save the simulation history. It can also return messages to the simulator for display. Messages received from the model managers are prefixed with the name of the model manager.

Log output of the Model Manager is incorporated directly into the `ncsim` log output (`ncsim.log` and `stdout`) by default. This logging output includes not only the special messages which `ncsim` normally incorporates, but also output from internal model output statements like `$display` and `$monitor`. You can control the log output with model

manager options. For example, you can suppress the model manager output with the `+ampnolog` model manager option, or you can redirect the output to another file using the `+amplogfile filename` option.

## Simulating OMI Models Controlled by C++ Model Managers

In order to simulate OMI models that are controlled by C++ model managers (SPW, for example), you must create a new elaborator and simulator that is linked with a C++ compiler rather than with a C compiler. This ensures correct initialization of memory management (constructors and destructors), and file I/O. Failure to do this results in a message that looks something like this:

```
% ncelab omitest_tb
ncelab: v2.2.(d5): (c) Copyright 1995-1999 Cadence Design Systems, Inc.
ncelab: *F,OMILDD: Could not load OMI Model Manager /ModelManager/libspwMM_c40.so
(1d.so.1: ncelab: fatal: relocation error: symbol not found: _pure_error_:
referenced in ./ModelManager/libspwMM_c40.so).
```

To link the C++ versions:

1. Log into a machine that has a C++ compiler installed on it. The C++ compiler should be the same compiler that was used to compile the model manager.
2. Create a directory to hold the new *ncelab* and *ncsim* executables.
3. Go to the directory that you just created.
4. Copy the example Makefile in the installation hierarchy into the directory. The example Makefile is located in:

*install\_directory/tools/inca/files/Makefile.nc*

5. Edit the Makefile. At the top of the Makefile, set the `CCC` macro to the location of the C++ compiler, if necessary. On Solaris, the change might look like this:

`CCC=/usr1/opt/SUNWspro/SC4.2/bin/CC`

6. Edit the `CCC_OBJECTS` macro to include the C++ object files.
7. Type `make ccc_static` to execute the branch of the makefile that builds the executables.

# Cosimulation with NC-Verilog and Quickturn

---

Verilog HDL designs can include models that simulate in conjunction with Quickturn Emulation Systems. You can use the NC-Verilog simulator as a front-end and testbench simulator and use the Quickturn emulators to perform hardware emulation.

This chapter contains the following sections:

- [Cosimulation Software Overview](#)
- [Setting Up the Simulator for Cosimulation](#)
- [Simulating a Model with NC-Verilog and Quickturn](#)
- [Restrictions and Limitations](#)

## Cosimulation Software Overview

The software elements for cosimulation are:

- The NC-Verilog simulator.
- Quickturn Design Systems Q/Bridge emulation environment, which allows Quickturn emulators to work with the simulator.
- Cadence Model Manager for Quickturn™, which manages the interaction between the Quickturn emulation system and the simulator. With the Cadence Model Manager for Quickturn, you can simulate in an environment where, for example, the testbench in the simulator provides the stimulus to the design in a hardware emulator. Other environments simulate with both the design and the testbench in the same domain.

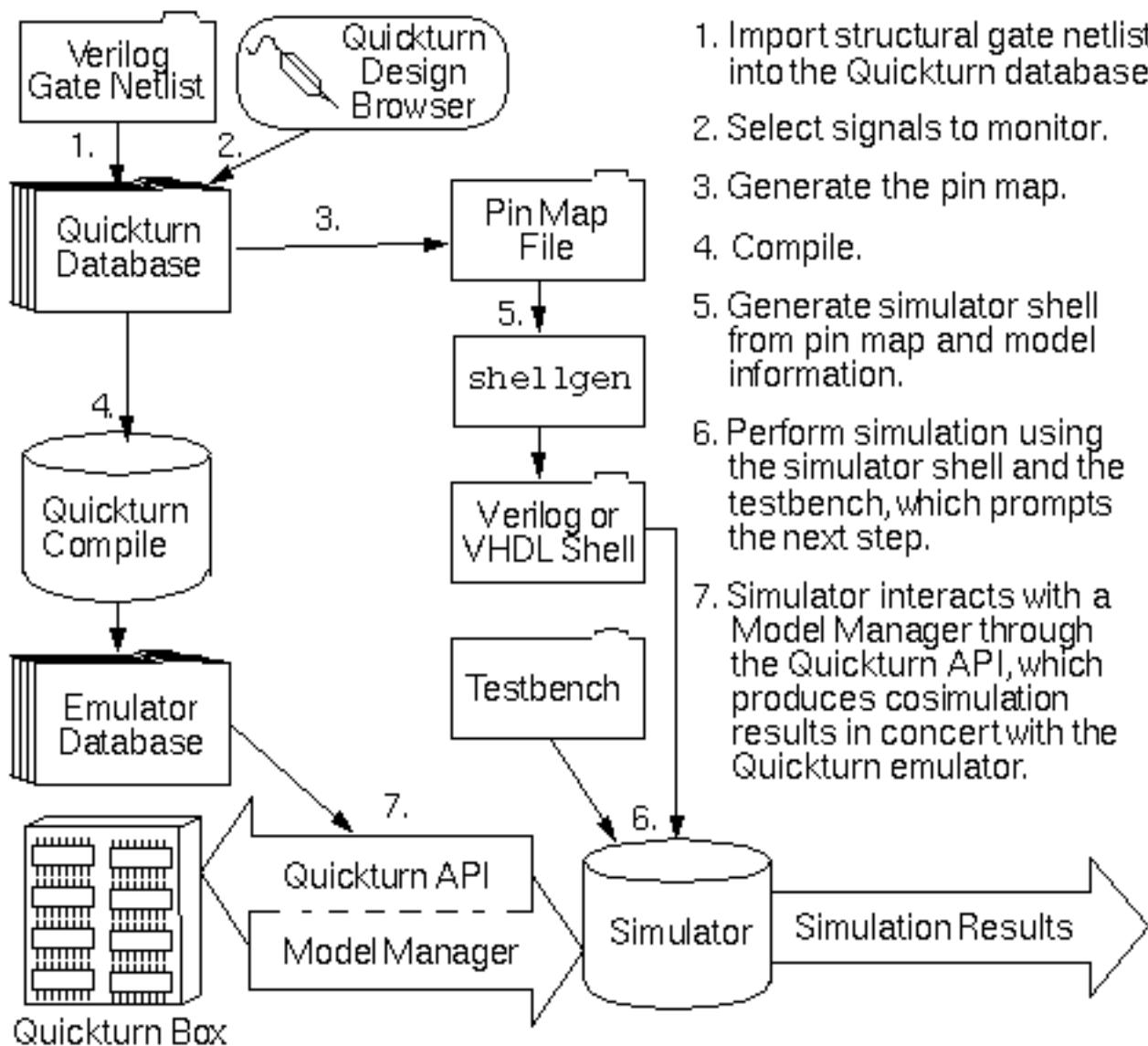
You can also have part of a design on the simulator and some instantiated part on the emulator. However, for performance reasons, Cadence recommends that you download the entire design to the emulator and leave only the testbench on the simulator.

## NC-Verilog Simulator Help

### Cosimulation with NC-Verilog and Quickturn

- The *shellgen* utility, which loads the Cadence Model Manager for Quickturn and creates the shell file using a pin map.

The following figure shows an overview of simulation with Quickturn.



## Setting Up the Simulator for Cosimulation

Before you can cosimulate, you must complete the preparation phases that are described in the following sections:

- [“Accessing Quickturn Integration” on page 969](#)
- [“Creating the Gate-Level Netlist” on page 969](#)
- [“Generating a Quickturn Emulator Database and a Pin Map” on page 970](#)
- [“Generating the Simulation Shell and Modifying the Testbench” on page 970](#)

### Accessing Quickturn Integration

To provide the simulator with access to Quickturn integration, perform the following steps:

1. Install Product 20600 using Cadence SoftLoad, which supplies the Quickturn Model Manager necessary for cosimulation.
2. In your `.cshrc` file, add the following paths to the Quickturn software and emulators:

```
setenv QT_HOME Quickturn_installed_tools  
set path = ($QT_HOME/bin $path)
```

(For Solaris):

```
setenv LD_LIBRARY_PATH $QT_HOME/lib/your_platform:$LD_LIBRARY_PATH
```

(For HP):

```
setenv $SHLIB_PATH $QT_HOME/lib/<your_platform>:$SHLIB_PATH
```

3. Install the Model Manager license using Cadence SoftLoad.

### Creating the Gate-Level Netlist

The Quickturn database requires gate-level input in either Verilog or Electronic Data Interchange Format (EDIF). Before you can import either a Verilog or VHDL design into the Quickturn database, you must synthesize the design into a Verilog gate-level or EDIF netlist.

**Note:** If you plan to synthesize the design, first verify that library primitives are supported by Quickturn. Then make sure that you have set up a common library between the synthesis netlist and `questCompile`.

## Generating a Quickturn Emulator Database and a Pin Map

To generate an emulator database and a pin map file, refer to the “Compiling IC Designs” chapter of the *Q/Bridge User’s Guide* from Quickturn Design Systems.

## Generating the Simulation Shell and Modifying the Testbench

At the workstation that contains the simulator software and the testbench design, perform these steps:

1. Generate the simulation shell using the *shellgen* utility. See “[shellgen](#)” on page 934 for details on using the *shellgen* utility and for an example.
2. Modify your testbench by instantiating the component from the shell into the testbench.

**Note:** These modifications pertain to generating a shell for the Cadence Model Manager for Quickturn’s pseudo event mode option. For more information, see “[Event Mode](#)” on page 971 and “[Clocked Mode](#)” on page 972.

The order of I/O pins in the shell file is determined by the order of the I/O pins in the pin map file. If the order of the I/O pins in the shell file and its instantiation in the testbench do not match, modify the instantiation or the pin map file. If you modify the pin map file, you must compile a new emulation database.

**Note:** Do not modify the order of the I/O pins in the shell file, or signals will not be correctly connected between the simulator and emulator.

## Cadence Model Manager for Quickturn Command-Line Plus Options

If you are using the Cadence Model Manager for Quickturn with the *shellgen* utility, you can specify the following plus options on the *shellgen* command line.

---

Plus option	Description
+qt_model =modelname	Specifies the name of the model for which the shell is being generated.  This option is required for shell generation for Quickturn.
+qt_pin_map =filename	Specifies the path to the pin map file in the Q/Bridge database directory. The Quickturn model manager uses this file to generate a shell with the shell generator.  This option is required for shell generation for Quickturn.

## NC-Verilog Simulator Help

### Cosimulation with NC-Verilog and Quickturn

---

<b>Plus option</b>	<b>Description</b>
+qt_mode= <i>mode</i>	Specifies how the model manager relays stimulus from the simulation to the emulator. The mode parameter can be one of the following:  event                   clockedDelayed clocked                clockedPosDelayed clockedPos            clockedNegDelayed clockedNeg
	For more information about these modes, see <a href="#">“Quickturn Modes for the qt_mode Option” on page 971</a> .
+qt_lib= <i>path</i>	Specifies the path to the Quickturn integration Library. The default is libqteapi.
+qt_emulator =address	Specifies the host name or address of the Quickturn emulator. The default is emulator.
+qt_qbridge =address	Specifies the host name or address of the NT server of Q/Bridge. The default is qbridge. If your Q/Bridge system does not use an NT Server, specify the host name or address of the Quickturn emulator.
+qt_strobedelay =[1...15]	Specifies a number from 1 to 15 that the model manager uses to control how long the NT server waits before it strobes the output of the emulator. The default is 15 time units.
+qt_outputdelay =value	Specifies the delay that the model manager uses to control the update of the model's output. You must specify an integer that is in the simulator's precision. The minimum value is 1. The default is 1.

---

### **Quickturn Modes for the qt\_mode Option**

The model manager operates in either event mode or clocked mode.

#### **Event Mode**

In event mode, the model manager collects any input changes within a specified time range and downloads them at the end of the cycle. After a specified simulation delay (using the +qt\_outputdelay plus option), the model manager pushes only the changed output signals into simulation.

When the model manager updates the status of the outputs from the emulator for the first time, the output ports transition from an X value to a known state. The model manager subsequently updates only the outputs that have changed since the previous update. The same value is never driven into an output port.

### Clocked Mode

In clocked mode, the shell generator adds an input pin to the shell, called `SYSTEMCLOCK`, at the end of the port list. The test fixture drives `SYSTEMCLOCK`, which in turn controls the inputs and outputs from the Quickturn emulator. The emulator drives the output values into the simulation at the opposite edge of the `SYSTEMCLOCK` signal.

The clocked mode options are as follows:

<b>qt_mode option</b>	<b>Description</b>
<code>event</code>	This is the default. Specifies the current edge of the <code>SYSTEMCLOCK</code> signal that the Model Manager uses to download the new input values to the emulator. The emulator drives the output values into the simulation at the opposite edge of the <code>SYSTEMCLOCK</code> signal.
<code>clocked</code> or <code>clockedPos</code>	Downloads the new input values to the emulator when the <code>SYSTEMCLOCK</code> signal makes a 0 to 1 transition.
<code>clockedNeg</code>	Downloads the new input values to the emulator when the <code>SYSTEMCLOCK</code> signal makes a 1 to 0 transition.
<code>clockedDelayed</code> or <code>clockedPosDelayed</code>	After waiting for a delay specified in the <code>+qt_ouputdelay</code> plus option, downloads the new input values to the emulator when the <code>SYSTEMCLOCK</code> signal makes a 0 to 1 transition.
<code>clockedNegDelayed</code>	After waiting for a delay specified in the <code>+qt_ouputdelay</code> plus option, downloads the new input values to the emulator when the <code>SYSTEMCLOCK</code> signal makes a 1 to 0 transition.

### Specifying Cadence Model Manager for Quickturn Options at Simulation Time

You can pass the following Cadence Model Manager for Quickturn options at simulation time, instead of specifying them to the shell generator:

- `qt_lib`
- `qt_emulator`

- qt\_qbridge
- qt\_strobedelay
- qt\_outputdelay

To pass Cadence Model Manager for Quickturn options at simulation time, use the option with the `ncsim` command. For example:

```
% ncsim +qt_lib=/mylib/quickturnlib top
```

Options specified on the `ncsim` command line override options specified to the shell generator.

## omi Command

The `omi` command lets you display information about the Quickturn model manager and instances controlled by the model manager and to pass model manager run-time commands to the model manager.

Quickturn supports only instance-specific commands to the model manager. Any `omi` command must be instance-specific.

See “[omi](#)” on page 568 for details on the `omi` command.

## \$omiCommand System Task

You can access the emulator memory by using the `$omiCommand` system task, which has the following syntax:

```
$omiCommand(<instance>, "<argument list>");  
<instance>  
:= name of the emulator shell  
  
<argument list>  
:= "listmem {CMM | LMM}"  
| |= "readmem <QTmemory> <filename> <radix> <start> <end>"  
| |= "writemem <QTmemory> <filename> <radix>"  
  
<QTmemory>  
:= the memory object in the source file
```

```
<filename>
  := name of the file that is loaded into memory (readmem)
  ||= name of the file into which memory is loaded (writemem)

<radix>
  := 2 (binary) | 8 (octal) | 10 (decimal) | 16 (hexadecimal)

<start>
  := starting memory address

<end>
  := ending memory address
```

For information about the format of the memory file, see the *Q/Bridge User Guide*.

The following examples show how to use the \$omiCommand system task.

```
$omiCommand(testFix.DUT, "listmem CMM");
```

This command displays a list of different memory contents in the CMM design type.

```
$omiCommand(testFix.DUT, "readmem u1.mem file.mem 16 0 100");
```

This example loads the contents of the file into memory from address 0 to address 100, hexadecimal.

```
$omiCommand(testFix.DUT, "writemem u1.mem file.mem 16");
```

This example loads the contents of memory into the specified file in hexadecimal format.

## Simulating a Model with NC-Verilog and Quickturn

When you invoke *ncsim* to simulate a model with Quickturn, the simulator recognizes the model as a Quickturn model and invokes the Cadence Quickturn Model Manager to initialize and control the Quickturn emulator.

To simulate a model, you must have these files:

- Testbench
- Simulation shell file
- Quickturn database

## Restrictions and Limitations

The following restrictions and limitations apply to the NC-Verilog simulator and Quickturn cosimulation:

- You can use all Tcl commands on ports. You can use all Tcl commands on viewports except assignment commands.
- The NC-Verilog simulator does not support the `save` and `restore` commands when a Quickturn model is integrated into the design.

---

## **Basics of Tcl**

---

This topic contains the following sections:

- [Overview](#)
- [Tcl Basics](#)
- [Extensions to Tcl](#)
- [Enabling Tk in the NC-Verilog Simulator](#)

### **Overview**

The *ncsim* simulator uses the Tool Command Language (Tcl) to control the execution of a simulation. Cadence has extended the functionality of the Tcl command interpreter so that it understands a number of new commands and some new syntax.

This appendix provides some basic information on Tcl syntax and information on extensions to the Tcl syntax.

For a complete description of Tcl, refer to *Tcl and the Tk Toolkit* by John K. Ousterhout (Addison-Wesley, Reading, MA, 1994).

There are also several Web sites that provide information on Tcl. The following sites are particularly useful:

- <http://www.elf.org>
- <http://dev.scriptics.com/>
- <http://www.tcltk.com/>

## Tcl Basics

This section provides basic information on Tcl.

### Tcl Variables

A simple Tcl variable has a name and a value. Both the name and the value can be arbitrary strings of characters. Variable names are case sensitive.

The value of a variable is always stored as a character string. Tcl variables can be used to represent many things, such as integers, real numbers, names, lists, and Tcl scripts, but they are always stored as strings. For example, if you use the `set` command to assign a value to a variable, as in

```
ncsim> set a 140  
140
```

it is critical to understand that the value of `a` is the character string `140`, not the integer `140`.

Variables are created automatically when they are assigned values.

### Variable Substitution

Placing an unquoted dollar sign character (`$`) in front of a variable name triggers variable substitution. All characters following the `$` that are letters, digits, or underscores are treated as a variable name. The `$` and the name are replaced in the word by the value of the variable.

In the following example, `$a` is replaced with the value of `a`, the character string `140`. This value is then assigned to the variable `b`.

```
ncsim> set a 140  
140  
ncsim> set b $a  
140
```

### Tcl Commands

A Tcl *command* consists of one or more *words*. The first word is the name of the command and additional words are arguments to that command. Words are separated by spaces or tabs. The Tcl interpreter prints the result of the command.

## The **set** Command

The **set** command is used to create, read, and modify variables. This command takes two arguments, and assigns the value of the second argument to the first argument.

```
ncsim> set a 24  
24  
ncsim> set 42 b  
b
```

In the second example, a variable named **42** is created, and is assigned the value **b**.

## The **expr** Command

The **expr** command is used to evaluate arithmetic expressions. The argument must be an expression.

The **expr** command returns the string value of the computed expression.

```
ncsim> expr 24/3.2  
7.5  
ncsim> expr (2+3) * 3  
15
```

## Command Scripts

Tcl commands can be combined into scripts by separating the commands with either a newline or a semicolon.

```
set a 20  
set b 40  
40  
ncsim> set a 20; set b 40  
40
```

In this example, **a** is assigned the string value of 20, and **b** is assigned 40.

When executing a command script, the command interpreter only prints the result of the last command in the script.

## Command Substitution

In addition to variable substitution (see “[Variable Substitution](#)” on page 977), Tcl also allows command substitution, which causes part or all of a word to be replaced with the result of a Tcl command.

Enclose the command in brackets to invoke command substitution.

```
ncsim> set inches 20  
20  
ncsim> set cm [expr $inches*2.54]  
50.8
```

The characters enclosed in brackets must be a valid Tcl script. The brackets and all of the characters between them are replaced with the result of the script.

## Backslash Substitution

Backslash substitution is used in Tcl to insert special characters, such as newlines, [, and \$, without them being treated specially by the Tcl interpreter.

In the following example, the value 24 is assigned to the variable a. In the second command, \$a is replaced by the value 24, and this is assigned to b. In the third command, however, the ‘\’ prevents the \$ from being treated specially (that is, triggering variable substitution), and therefore the string value of \$a is assigned to b.

```
ncsim> set a 24  
24  
ncsim> set b $a  
24  
ncsim> set b \$a  
$a
```

## Value Substitution

Using the value of an object in a command is very common:

```
ncsim> set a [value w]
```

The Tcl interpreter includes a shorthand notation, using the pound sign (#), for this:

```
ncsim> set a #w
```

This notation is completely analogous to the \$ variable substitution that is standard in Tcl (see “[Variable Substitution](#)” on page 977), although a format specifier capability has been added.

To control the format of the value substitution, a format character preceded by a percent sign can follow the #. For example,

```
ncsim> value w  
8'b11111111  
ncsim> set a #%xw  
8'hff
```

Because all format specifiers are one character long, no special syntax is needed to separate the format specifier from the HDL name.

If the substituted value is to be used as part of a command-line argument, braces can be used to delineate the extent of the HDL name in the same way that they are used by Tcl (and the c-shell) to delineate variable names. For example, the following command sets x to the value of w with 10 appended.

```
ncsim> set x #%x{w}10  
8'ff10
```

#%x{*name*} is syntactically equivalent to [value %x *name*].

## Quoting Words in a Command

There are two ways to *quote* words in a command: with double quotes (" ") or with braces ({}).

The following example shows the difference in the output between using quotation marks and braces.

```
ncsim> set a 24  
ncsim> set msg "The value of \$b = [expr $a/2]"  
The value of $b = 12  
ncsim> set msg {The value of \$b = [expr $a/2]}  
The value of \$b = [expr $a/2]
```

## Using Double Quotes

If you enclose a word in a command in double quotes:

- Word and command separators are disabled.
- Spaces, tabs, newlines, and semicolons are treated as ordinary characters within the word.
- Variable substitution, command substitution, and backslash substitution all occur as usual inside the double quotes.

For example, the following script sets `msg` to a string containing the name of a variable, and the value of a numeric expression:

```
ncsim> set msg "The value of \$b = [expr $a/2]"  
The value of $b = 12
```

## Using Braces

If you enclose a word in a command in braces ( {} ):

- All special characters lose their meaning.
- All characters, including spaces, tabs, newlines and semicolons, are treated as ordinary characters.
- No substitutions are performed.

In the following example, the value of `msg` is the entire string, verbatim, that is enclosed by the braces.

```
ncsim> set msg {The value of \$b = [expr $a/2]}  
The value of \$b = [expr $a/2]
```

Use braces to prevent the immediate processing of special characters by the Tcl parser. This is called *deferrred evaluation*. Special characters are passed to the command procedure as part of its argument. The command procedure then processes the special characters itself, often by passing the argument back to the Tcl interpreter for evaluation.

In the following example, the `proc` command is used to create a procedure that adds two numbers:

```
ncsim> proc add {a1 a2} {  
> set sum [expr $a1+$a2];return $sum  
> }  
ncsim> add 3 4  
7
```

Because the body of the procedure is enclosed in braces, it is passed verbatim to `proc`. The value of the variables `a1` and `a2` is not substituted when the `proc` command is parsed. This is necessary because a different value must be substituted for these variables each time the procedure is invoked.

## Extensions to Tcl

The functionality of the Tcl expression evaluator has been extended to handle types and operators of the Verilog and VHDL hardware description languages.

The Tcl command interpreter has also been enhanced so that it understands some new syntax and new commands. See [Chapter 12, “Using the Tcl Command-Line Interface,”](#) for descriptions of the new ncsim-specific commands.

## Expression Evaluation

Tcl has a built-in `expr` command that parses and evaluates numeric expressions. This command handles the standard arithmetic and logical operators on operands which are of integer, real, and string types. This facility has been enhanced for both Verilog and for VHDL to handle types and operators of these languages.

## Verilog Expressions

Verilog has one main data type (a four-state logic vector) and the operators on this type are clearly defined in the language. This type and its operators have been incorporated into the expression evaluator so that just about any expression appearing in Verilog code can be mimicked in Tcl.

The new data type added to the Tcl parser to support the evaluation of Verilog expressions is Verilog Register. The Verilog Register type is an unsigned vector type of arbitrary bit size whose values are Verilog literals, such as '`b1001`', `16'h8fff0`, and `8'bxxxxxxxx01`.

Any expression that has an operand in this form is treated as a Verilog expression, and the expression evaluator follows the rules of Verilog literals and operators when evaluating it.

```
ncsim> expr 'h1f + 42
32'b1001001
ncsim> value reg
8'b11111111
ncsim> expr #reg & ~32'h1f
32'b11100000
```

The format of the value returned from the `expr` command is controlled by the `vlog_format` variable.

## Operators Added to Tcl

The Tcl expression parser already supports most of the logical and arithmetic operator symbols of Verilog. The following operators have been added:

**Table A-1 Binary Operator Extensions**

Operator	Function
<code>==</code>	Case equality. See “ <a href="#">Equality Operators</a> ” on page 984 for more information on equality operators.
<code>!=</code>	Case inequality
<code>~^</code>	Bit-wise XNOR
<code>^~</code>	Another symbol for bit-wise XNOR

**Table A-2 Unary Operator Extensions**

Operator	Function
<code>&amp;</code>	Reduction AND
<code>~&amp;</code>	Reduction NAND
<code> </code>	Reduction OR
<code>~ </code>	Reduction NOR
<code>^</code>	Reduction XOR
<code>~^</code>	Reduction XNOR
<code>^~</code>	Another symbol for reduction XNOR

## Concatenation Operators

Because native Tcl syntax uses braces for suppressing substitution, the Tcl interpreter is unable to support braces for use as the Verilog concatenation operator. Instead, you can use a function called `vcat`, which takes an arbitrary number of arguments and returns a Verilog Register value which is the concatenation of all the bits in all the arguments.

```
ncsim> expr vcat(3'b101, 5'b1, #r)
16'b101000011111111
```

You can also use the `vcat` function to input a string as a Verilog value, as shown in the following example:

```
ncsim> expr vcat("Hello")
40'b100100001100101011011000110110001101111
```

You can use a function called `vrep` to repeat concatenation. This function takes an integer repetition count and a Verilog value as arguments and returns a value whose bit pattern is that of the second argument repeated the number of times specified by the first argument:

```
ncsim> expr vrep(3, 3'b101)
9'b101101101
```

## Logical AND and OR Operators

Tcl defines the logical AND and OR operators to be short-circuiting. That is, if the first operand is sufficient to determine the result of the operator, then the second operand is not evaluated. This conflicts with Verilog in that both operands are needed to determine whether the Verilog version of the operator should be performed, and the bit size of the result.

Because of this, these operators will not behave exactly as they do in Verilog, but will always return an integer, either 1 or 0. If the result of the expression contains `x` or `z` bits, these are treated as zeroes.

```
ncsim> expr 2'b11 || 5'b0
1
ncsim> expr 'bx || 2'b1
1
ncsim> expr 2'b01 && 5'b10
1
ncsim> expr 'bx && 2'b1
0
```

## Equality Operators

The single-equality operator (`=`) is used for assignment in Verilog. In Tcl, this operator is a logical comparison operator. In a Tcl expression, `=` is the same as the Verilog logical comparison operator (`==`). The following two expressions are the same:

```
{ #top.load = 1'b1 }
{ #top.load == 1'b1 }
```

These operators return the unknown value (x) if either operand is unknown. For example, the following expression returns an unknown result because one of the operands is unknown:

```
{ #top.load == 1'bx }
```

For the case equality operator ( `==` ) and the case inequality operator ( `!=` ), bits that are unknown are included in the comparison, and the result of the expression is always 1 (true) or 0 (false).

In a conditional expression, an unknown result is treated as false. For example, in the following `stop` command, the expression returns an unknown result, and is, therefore, false. This conditional breakpoint will not trigger when the signal `top.load` has the value `x`.

```
ncsim> stop -create -condition {#top.load == 1'bx}
```

To set a breakpoint that will stop when the value is `x`, use the case equality operator, as follows:

```
ncsim> stop -create -condition {#top.load === 1'bx}
```

## Conversion Functions

The functions `vhex`, `voct`, `vdec` and `vbin` convert a Verilog value or an integer to a Verilog literal in the corresponding format. The result type of these functions is a string. These functions are provided simply for base conversion and output formatting.

```
ncsim> expr vhex('b111100101)
32'h1e5
ncsim> expr voct(9'b111100101)
9'o745
ncsim> expr vdec(9'o745)
9'd485
ncsim> expr vbin(400 + 80 + 5)
32'b111100101
```

To turn a bit pattern into a string, the function `vstr` takes a bit pattern argument and returns the equivalent string:

```
ncsim> expr vstr(16'b100100001101001)
Hi
```

## VHDL Expressions

In VHDL, unlike in Verilog, there are many data types, both predefined and user-defined, and the user can redefine the operator functions for these types. Because it is not possible to build all possible VHDL expressions into Tcl, the most common data types and the predefined versions of the operators for these types that are contained in standard packages have been built into the Tcl interpreter.

The types STD. INTEGER, STD. REAL, and STD. STRING are handled in basic Tcl. The expression evaluator has been enhanced to handle enumeration values and vectors of enumeration values. These are critical because they are used to represent logic vectors. The enhancements allow the expression evaluator to handle enumeration literals and vector literals and the predefined operators that VHDL provides for them.

Because logic vectors are the basis of most simulations, and because the predefined operators in VHDL for arrays of enumeration types are not sufficient to make them useful as logic types, a package that defines a logic type and its operators is generally used. Since this is most commonly IEEE. STD\_LOGIC\_1164, the operators that are defined in this package have also been built into the Tcl expression evaluator.

### Enumeration Literals

In basic Tcl, the type of a literal can be determined directly. Integers and reals contain only digits, decimal points, and other well-defined characters. Strings are always enclosed in double-quotes. Verilog literals have a specific format that includes a tick ( ` ) followed by a radix character. In VHDL, a word that is not followed by parentheses (as for a function call) is treated as a VHDL enumeration literal. A character that is enclosed in single quotes is also treated as an enumeration literal. Anything else is an invocation of a math function (such as sin(1)) or a syntax error.

To be able to use an enumeration literal, the expression evaluator must also be able to determine the type declaration to which the literal belongs. In order to tell the expression evaluator what type a literal is, you can include the name of the type with the literal. The format is `type:literal`, where `type` is a local unit path name to the enumeration type, and `literal` is the enumeration literal.

Use the %e format specifier on the `value` command to generate a fully qualified enumeration literal. Without a format specifier, an unqualified enumeration literal will result, as shown in the following example.

```
ncsim> value color
green
ncsim> value %e color
@mylib.color_pkg:colors:green
```

This syntax assumes that the object called `color` is of type `colors`, which is declared in a package called `color_pkg` in the design library `mylib`.

For types that are declared in architectures, the syntax is:

```
ncsim> value %e x  
@mylib.testbench behav :my_logic:logic_1
```

where `testbench` is the entity, `behav` is the architecture, and `my_logic` is the type.

You almost never have to type this format because:

- If a binary operation is being performed, the expression evaluator only needs to know the type of one operand. It assumes that the other operand is of the same type.
- When the value of an HDL object is substituted using value substitution, the fully typed format is used. That is, the expression evaluator understands an expression such as `{#color = green}`, where `color` is an enumeration type object, and `green` is a literal of that type.

The operators for enumeration types are those that are implicitly defined in VHDL for all enumeration types. These are the equality and relational operators. Addition and subtraction of two enumeration literals, or one enumeration literal and one integer is also supported. For these operators, operands that are enumeration literals are converted to their position value, the operation is performed as for integers, the result is taken as a position value, and this result is then converted back to the corresponding enumeration literal. For example:

```
ncsim> expr green + 1  
ncsim: *E,TCLERR: cannot determine enumeration type for "green" (operator: +).  
ncsim> expr @mylib.color_pkg:colors:green + 1  
@mylib.color_pkg:colors:blue
```

## Enumeration Vectors

Enumeration vectors that can appear in VHDL as strings are handled in the same way as enumeration literals. Strings can be qualified with a type name, just as enumeration literals can. If the type of a string can be deduced from the context, it does not have to be qualified. For example:

```
ncsim> value my_bus  
"11001100"  
ncsim> value %e my_bus  
@std.bit:"11001100"  
ncsim> expr #my_bus = \"11001100\"  
@std.standard:boolean:true
```

In these examples, although `my_bus` is of type `bit_vector`, the qualifying type is the element type `bit`. When Tcl generates a qualified enumeration literal, it uses the name of the base enumeration type. If you enter a qualified literal, the qualifying type can be any subtype of an enumeration type or an array type whose element type is an enumeration. Either of these is equivalent to using the enumeration base type directly.

In the last command shown above, the double-quotes around the vector literal are required. The backslashes are necessary so that the quotes are not stripped by the Tcl parser. Alternatively, the expression can be enclosed in braces so that the backslashes are not necessary. For example:

```
ncsim> expr {#my_bus = "11001100"}  
@std.standard:boolean:true
```

The result of the `expr` command is a fully qualified literal. If the result is a VHDL enumeration or vector type, it includes the type qualification. In the example above, the result of the equality operator is of type `STD.STANDARD:BOOLEAN`. Because the operands are VHDL-specific types, the VHDL definition of the operator is used. The LRM defines this operator to return the VHDL boolean type.

Operators on enumeration vector types are those that are implicitly defined in VHDL for string types. These are the equality operators, relational operators, and the concatenation operator.

The concatenation operator symbol in VHDL is an ampersand (`&`). This is the same as the Verilog bitwise-and operator symbol. This symbol represents both operations. The expression evaluator looks at the types of its operands to determine which operation to perform.

The precedence of the `&` operator in VHDL is higher than that of the `&` operator in Verilog, but the expression evaluator cannot distinguish the two at the time it needs to know the operator precedence. Therefore, the VHDL `&` operator has the same precedence as the Verilog `&` operator. If you use this operator in a complex expression, use parentheses to enforce the desired precedence.

## Standard Logic Type

Additional support is provided for the logic type defined in the IEEE 1164 standard packages. The base enumeration type that this package defines is called `STD_ULOGIC`. It is a nine-state logic type. The packages that define this type and its operators are `STD_LOGIC_1164` and `STD_LOGIC_ARITH`. There is a resolved subtype of `STD_ULOGIC` that is called `STD_LOGIC`, so you can use either name to qualify a literal of this type.

The operators that are defined for operands of type `STD_ULOGIC` are those that are defined in the `IEEE.STD_LOGIC_1164` and `IEEE.STD_LOGIC_ARITH` packages. These operators

## NC-Verilog Simulator Help

### Basics of Tcl

---

have definitions other than the standard VHDL definition for one-dimensional discrete arrays. These operations are described in the next section. Here are some examples:

```
ncsim> expr @ieee.std_logic_1164:std_logic:'X'  
@ieee.std_logic_1164:std_ulogic:'X'  
ncsim> value %e my_bus  
@ieee.std_logic_1164:std_ulogic:"00001010"  
ncsim> expr #my_bus + 10  
@ieee.std_logic_1164:std_ulogic:"00010100"  
ncsim> expr #my_bus + 0x1f  
@ieee.std_logic_1164:std_ulogic:"00101001"
```

### VHDL operators

The following table shows the operators that have been added to support VHDL. In the table, the term *any VHDL type* means "any enumeration literal or vector type". A specific type such as *bit* or *boolean* refers to a literal of this type or to a vector with this as its element type. In all cases, an operand that is an enumeration literal is treated the same as a vector of length 1.

For precise definitions of the operators, see the VHDL Language Reference Manual or the source code for the IEEE standard logic packages.

Operation	Infix Symbol	Left operand type	Right operand type	Result type
BITWISE AND	and	Bit, boolean, or std_ulogic, any length	same type, same length	same type, same length
BITWISE NAND	nand			
BITWISE OR	or			
BITWISE NOR	nor			
BITWISE XOR	xor			
BITWISE XNOR	xnor			
BITWISE NOT	not	N/A	bit / boolean / std_ulogic, any length	same type, same length

## NC-Verilog Simulator Help

### Basics of Tcl

---

<b>Operation</b>	<b>Infix Symbol</b>	<b>Left operand type</b>	<b>Right operand type</b>	<b>Result type</b>
PLUS MINUS	+ -	any VHDL type, length 1	same type, length 1	same type, length 1
		any VHDL type, length 1	integer	same VHDL type, length 1
		integer	any VHDL type, length 1	same VHDL type, length 1
PLUS MINUS MULTIPLY DIVIDE	+ - * /	std_ulogic, any length	std_ulogic, any length	std_ulogic, length of longer operand
		std_ulogic, any length	integer	std_ulogic, length of left operand
		integer	std_ulogic, any length	std_ulogic, length of right operand
LESS THAN LESS OR EQUAL GREATER THAN GREATER OR EQUAL EQUAL TO NOT EQUAL TO	< <= > >= =, == /=, !=	any VHDL type, any length	same type, same length	boolean, length 1
		std_ulogic, any length	integer	
		integer	std_ulogic, any length	
LOGICAL SHIFT LEFT LOGICAL SHIFT RIGHT ROTATE LEFT ROTATE RIGHT	sll srl rol ror	bit, boolean, or std_ulogic, any length	integer	same VHDL type, same length
ARITHMETIC SHIFT LEFT ARITHMETIC SHIFT RIGHT	sla sra	bit or boolean, any length	integer	same VHDL type, same length
CONCATENATION	&	any VHDL type, any length	same type, any length	same type, sum of operand's lengths
EXPONENTIATION	**	integer or real	integer	integer or real

Operation	Infix Symbol	Left operand type	Right operand type	Result type
ABSOLUTE VALUE	abs	N/A	integer or real	same type
MODULUS	mod, %	integer	integer	integer
REMAINDER	rem	integer	integer	integer

The bitwise logical operators and the logical shift operators that are shown in this table also exist in Verilog, but with different infix symbols. The symbols are not interchangeable. To get the VHDL version of the operator, you must use the VHDL infix symbol.

For the equality and modulus operators, you can use the Verilog symbol and the VHDL symbol interchangeably. The VHDL definition of these operators will be used if at least one of the operands is a VHDL enumeration type.

Arithmetic operations in which at least one of the operands is STD\_ULOGIC are defined in the standard logic package to be integer operations. In these cases, the STD\_ULOGIC operands are first converted to integers and the result is converted back to STD\_ULOGIC. The rules for this conversion are found in the standard logic package's `to_integer` function.

Relational operations in which one operand is STD\_ULOGIC and the other is an integer are also integer operations. The STD\_ULOGIC operand is first converted to an integer, and an integer comparison is performed. Relational operations on two STD\_ULOGIC vector values use the standard definitions for one-dimensional VHDL arrays.

## Tcl Functions for Type Conversion

Tcl has predefined functions for converting to and from integer and double (floating-point) types. These functions also work with VHDL and Verilog types.

■ `int(x)`

Converts its parameter to an integer.

- ❑ If the parameter is a floating-point value, the conversion is as defined by Tcl. The decimal portion is truncated.
- ❑ If the parameter is of STD\_ULOGIC type, the conversion is that of the function `to_integer` in the STD\_LOGIC\_ARITH package. If the parameter is of type BIT, it is converted as an unsigned binary number whose most significant bit is the left-most bit. For both of these VHDL types, the 32nd bit is a sign bit. Values that have fewer than 32 bits are unsigned. If the vector length is greater than 32, only the right-most 32 bits are used.

- ❑ If the parameter is a Verilog vector type, the conversion is as defined by Verilog for arithmetic operations, with x and z valued bits treated as zero bits, and with values longer than 32 bits truncated to the right-most 32 bits.
- **double(*x*)**  
Converts its parameter to a floating-point value.
  - ❑ For integers, the conversion is as defined in Tcl.
  - ❑ For VHDL types `BIT` and `STD_ULOGIC`, the parameter is first converted to an integer, as described above, and then this integer is converted to a floating-point value.
  - ❑ For Verilog values, x and z bits are changed to zero, and the resulting bit pattern is extended with zeroes or truncated to 64 bits. The bit pattern is interpreted as a 64-bit representation of a floating-point number.

Two new type conversion functions have also been added:

- **vlog(*x*)**  
Converts its parameter to the Verilog logic type.
  - ❑ If the parameter is a vector of type `STD_ULOGIC` or `BIT`, the conversion is the same that occurs across language boundaries in a mixed-language simulation.
  - ❑ If the parameter is an integer, the result is a Verilog vector of length 32.
  - ❑ If the parameter is a floating-point value, it is converted to a 64-bit Verilog value that represents the same floating-point value.
  - ❑ If the parameter is a string enclosed in quotes, it is converted to a Verilog logic vector representation of the ASCII codes for the characters in the string.
- **std\_logic(*x, size*)**  
Converts its first parameter to a `STD_ULOGIC` vector of the specified size.
  - ❑ If the parameter is a Verilog literal, the conversion is the same that occurs across language boundaries in a mixed-language simulation for nets without strength information on the Verilog side.
  - ❑ If the parameter is an integer, the conversion is that of the function `To_StdUlogicVector` defined in the `STD_LOGIC_ARITH` package.
  - ❑ If the parameter is a VHDL value, it must either be of type `BIT` or of type `STD_ULOGIC`. It is an error if the first parameter is a floating-point value.

If the given size is too small to contain the converted value, excess bits are truncated from the left end of the vector. If the size is larger than necessary, the resulting value is padded on the left with zeros (0) for integer conversion and with the unknown value (U) for Verilog and VHDL value conversion. If the given size is zero, the size of the converted value is the size of the value being converted (32 for integers). You cannot specify a size less than zero.

## Enabling Tk in the NC-Verilog Simulator

You can use Tk with the NC-Verilog simulator. In order to use it, you need a shared library version of Tk and the library of Tcl script files that comes with it. The version of Tk currently required by the simulator is 8.3.2. Later versions of Tk will not work.

Tk is not included as part of the simulator product, but is available on the internet. One site from which you can download version 8.3.2 is:

[ftp://ftp.scriptics.com/pub/tcl/tcl8\\_3/](ftp://ftp.scriptics.com/pub/tcl/tcl8_3/)

Download the following files:

- tcl8.3.2.tar.gz
- tk8.3.2.tar.gz

Untar these files in the same directory, and then execute the following commands:

1. cd *your\_tcl\_dir/tcl8.3.2/unix*
2. ./configure
3. make
4. cd ../../tk8.3.2/unix
5. ./configure --enable-shared

**Note:** This command could change. See the README file in the *tk8.3.2/unix* directory to verify that the command shown above is correct.

6. make

When you run *ncsim*, you can enable Tk with the following two commands:

*ncsim> set tk\_library your\_tcl\_dir/tk8.3.2/library*

(For Solaris) *ncsim> load your\_tcl\_dir/tk8.3.2/unix/libtk8.3.2.so*

(For HP) *ncsim> load your\_tcl\_dir/tk8.3.2/unix/libtk8.3.2.sl*

---

## SDF File Syntax

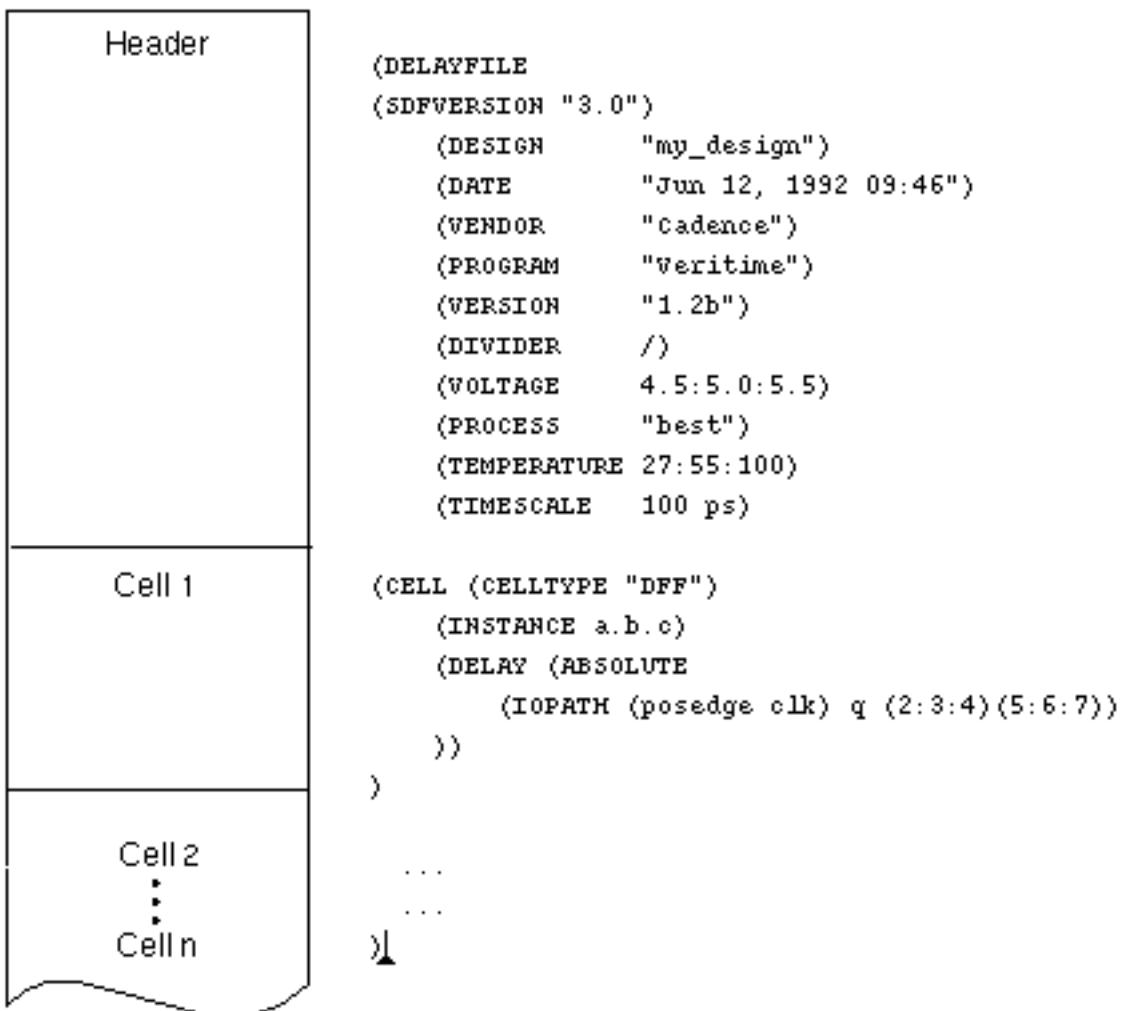
---

This topic describes the syntax of the SDF file. It contains the following sections:

- [Overview](#)
- [SDF File Conventions](#)
- [OVI Standard 3.0 SDF Keywords](#)
- [SDF File Keyword Constructs](#)
- [OVI SDF Specification Version Differences](#)
- [SDF File Examples](#)

## Overview

Every SDF file contains a header section followed by one or more cell entries. For each cell entry, you can specify delays, timing checks, and other constraints using a wide variety of keywords.



Although the NC-Verilog simulator can read an SDF file with all of the OVI SDF Standard keyword constructs, only a subset of the constructs are relevant to logic simulation. Constructs that are used by other tools (logic synthesis, layout, timing analysis, and so on) are ignored by NC-Verilog. No error message is generated, assuming that the constructs are syntactically correct.

The SDF annotator supports multiple versions of the OVI SDF specification. Because some constructs in one version may not available in other versions, you need to specify which

version you want to use in the `SDFVERSION` entry in the header section of the SDF file. If no `SDFVERSION` is specified, version 1.0 is used by default. In most cases, the difference between versions is minimal. If a construct that is not supported under the current version setting is encountered, you will receive a syntax error. See “[OVI SDF Specification Version Differences](#)” on page 1050 for details on some of the differences between versions.

“[OVI Standard 3.0 SDF Keywords](#)” on page 999 shows all of the constructs that are supported by the OVI SDF Specification, Version 3.0. Click on a keyword to go to the section describing the construct.

## SDF File Conventions

This section describes identifiers, character use, operators, and common expressions in SDF files.

### Identifiers

*Identifiers* are names for ports or nets, depending on the syntax. Identifiers can have up to 1024 alphanumeric characters. Special characters are permitted if the escape character (\) precedes the special character. Spaces are not allowed in identifiers.

You can specify hierarchical identifiers by placing the hierarchy divider character (. or /) in the identifier name.

Bit specifications can be placed at the end of identifiers with no spaces between the bit specifications and the identifier. Bit specifications are specified in square braces ([ ]). If the bit spec is a range, use a colon to separate the range, as shown in the following examples:

[4]            [3:31]            [15:0]

Edge identifiers are specified with the following names:

posedge        negedge        01        10        0z        z1        1z        z0

### Examples of Correct Identifiers

```
//From a language where square brackets indicate arrays
// parentheses indicate bit specs
AMUX\+BMUX
Cache_Row_\#4
mem_array\[0\:1023\]\(0\:15\
// Unescaped square brackets is a bit spec
pipe4\~done\&enb[3]
```

### **Examples of Incorrect Identifiers**

```
// Do not use Verilog style name escaping  
\AMUX+BMUX

// Spaces cannot be escaped  
PHASE\ LOCK\ DONE

// Do not use carriage return  
Ctl_Brk\

// Do not include bit specs within identifiers  
MEM[4:16]_BRK\+IDLE
```

The following table describes the characters you can use in the SDF file.

<b>Character Type</b>	<b>Characters</b>
Alphanumeric Characters	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z 0 1 2 3 4 5 6 7 8 9 _ (underscore)
Arithmetic Characters	+ (add), - (subtract), / (divide), * (multiply)
Bit-wise binary and	& (ampersand)
Bit-wise binary equivalence	^~ or ~^ (caret tilde or tilde caret)
Bit-wise binary exclusive or	^ (caret)
Bit-wise binary inclusive or	(vertical bar)
Bit-wise unary negation	~ (tilde)
Case equality operator	===(triple equals signs)
Case inequality operator	!= (exclamation equals equals)
Case inequality operator	!= (exclamation equals equals)
Comment Characters	// double slash for any single line /* begins comment text ending with */
Escape Character	\ (backslash)
Hierarchy dividers	. (period) or / (slash)
Left shift	<< (double left angle brackets)
Logical and	&& (double ampersand)

## NC-Verilog Simulator Help

### SDF File Syntax

---

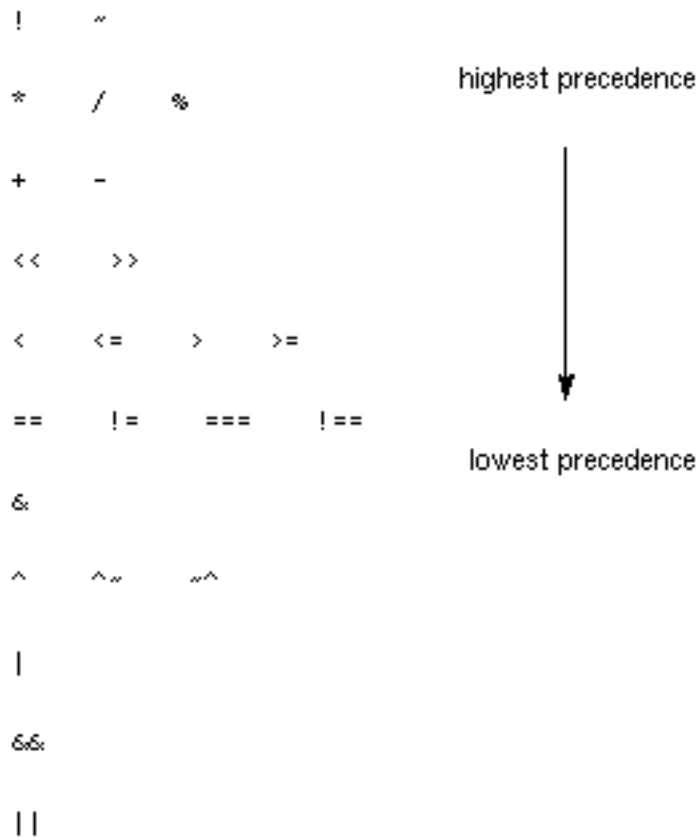
Character Type	Characters
Logical equality	<code>==</code> (double equals signs)
Logical inequality	<code>!=</code> (exclamation equals)
Logical negation	<code>!</code> (exclamation)
Logical or	<code>  </code> (double vertical bar)
Modulus	<code>%</code> (percentage)
Relational operators	<code>&gt;</code> (greater than), <code>&gt;=</code> (greater than or equal to), <code>&lt;</code> (less than), <code>&lt;=</code> (less than or equal to),
Right shift	<code>&gt;&gt;</code> (double right angle brackets)
Space	Space, tab, and new line.
Special Characters	<code>~ ! " # \$ % &amp; ' ( ) * + , - . / : ; &lt; = &gt; ? @ [ \ ] ^ ` {   }</code>
Unary Operators	<code>+ - ! ~ &amp; ~&amp;   ~  ^ ^~ ~^</code>
Binary Operators	<code>+ &lt; - &lt;= * &gt; / &gt;= % &amp; ==   != ^ === ^~ ! == ~^ &amp;&amp; &gt;&gt;    &lt;&lt;</code>

---

**Note:** The escape character (`\`) must precede a special character in a port or net identifier. If you use an escape character preceding a hierarchy divider character (`.` or `/`), the characters no longer divide the hierarchy.

## Operator Precedence

The following figure shows the order of precedence for SDF operators. Operators shown on the same row have the same precedence.



## OVI Standard 3.0 SDF Keywords

```
(DELAYFILE
  (SDFVERSION "sdf_version")
  (DESIGN "design_name")
  (DATE "date")
  (VENDOR "vendor_name")
  (PROGRAM "program_name")
  (VERSION "program_version")
  (DIVIDER hierarchy_divider)
  (VOLTAGE min:typ:max)
  (PROCESS "process_name")
```

## NC-Verilog Simulator Help

### SDF File Syntax

---

```
(TEMPERATURE min:typ:max)
(TIMESCALE time_scale)

(CELL (CELLTYPE ...))
  (INSTANCE ...)
  (INCLUDE ...)
  (DELAY
    (ABSOLUTE | INCREMENT
      (IOPATH ...))
    (COND ... (IOPATH ... {(RETAIN ...)}...))
    (CONDENSE ... (IOPATH ... { (RETAIN ...) } ...))
    (PORT ...)
    (INTERCONNECT ...)
    (DEVICE ...)
    ) // end ABSOLUTE or INCREMENT
    (PATHPULSE ...)
    (PATHPULSEPERCENT ...))
  ) // end DELAY

(TIMINGCHECK {COND ...})
  (SETUP ...)
  (HOLD ...)
  (SETUPHOLD ... { (SCOND ...) } { (CCOND ...) } )
  (RECOVERY ... { (SCOND ...) } { (CCOND ...) } )
  (REMOVAL ... { (SCOND ...) } { (CCOND ...) } )
  (RECREM ... { (SCOND ...) } { (CCOND ...) } )
  (SKEW ...)
  (WIDTH ...)
  (PERIOD ...)
  (NOCHANGE ...)
) // end TIMINGCHECK

(TIMINGENV
  (PATHCONSTRAINT ...)
  (PERIODCONSTRAINT ... (EXCEPTION (INSTANCE ...)))
  (SKEWCONSTRAINT ...)
  (SUM ...)
  (DIFF ...)
  (ARRIVAL ...)
  (DEPARTURE ...)
  (SLACK ...))
```

```
(WAVEFORM ...)  
) // end TIMINGENV  
) // end CELL  
) // end DELAYFILE
```

**Note:** The [NETDELAY](#) construct is not supported in version 3.0. The NC-Verilog annotator, however, supports multiple versions of the OVI specification. To use the NETDELAY construct, specify a version number that is lower than 3.0 with the [SDFVERSION](#) statement. For example,

```
(SDFVERSION "2.1")  
...  
...  
(NETDELAY ...)
```

## SDF File Keyword Constructs

### DELAYFILE Keyword

The DELAYFILE construct contains all header and CELL entries in an SDF file. Header entries must precede CELL entries.

You can specify any or all header entries, but they must be in the sequence shown in the following syntax:

```
(DELAYFILE  
{ (SDFVERSION "sdf_version") }  
{ (DESIGN "design_name") }  
{ (DATE "date") }  
{ (VENDOR "vendor_name") }  
{ (PROGRAM "program_name") }  
{ (VERSION "program_version") }  
{ (DIVIDER hierarchy_divider) }  
{ (VOLTAGE min:typ:max) }  
{ (PROCESS "process_name") }  
{ (TEMPERATURE min:typ:max) }  
{ (TIMESCALE time_scale) }  
(CELL cell_constructs)  
)
```

**Note:** NC-Verilog uses only the SDFVERSION, DIVIDER, and TIMESCALE header keywords.

## NC-Verilog Simulator Help

### SDF File Syntax

---

The following table describes the SDF file header keywords and arguments.

<b>Keyword</b>	<b>Keyword Argument</b>	<b>Description</b>
SDFVERSION	" <i>sdf_version</i> "	A string specified in quotation marks that specifies the SDF software version number.
DESIGN	" <i>design_name</i> "	A string specified in quotation marks that specifies the name of the design.
DATE	" <i>date</i> "	A string specified in quotation marks that specifies the date and time when SDF was generated.
VENDOR	" <i>vendor_name</i> "	A string specified in quotation marks that specifies the name of the vendor whose tools generated the SDF file.
PROGRAM	" <i>program_name</i> "	A string specified in quotation marks that specifies the name of the program used to generate the SDF file.
VERSION	" <i>program_version</i> "	A string specified in quotation marks that specifies the program version number used to generate the SDF file.
DIVIDER	<i>hierarchy_divider</i>	The hierarchical path divider your program is using. This can be either the period (.), which is the default, or the slash (/).
VOLTAGE	<i>min:typ:max</i>	Three values ( <i>min:typ:max</i> ) that specify the operating voltage (in volts) of the design.
PROCESS	" <i>process_name</i> "	A string specified in quotation marks that specifies the process operating envelope.
TEMPERATURE	<i>min:typ:max</i>	Three values ( <i>min:typ:max</i> ) that specify the operating ambient temperature(s) of the design in centigrade degrees.

## NC-Verilog Simulator Help

### SDF File Syntax

---

Keyword	Keyword Argument	Description
TIMESCALE	<i>time_scale</i>	A value followed by a time specification, such as 100 ps for 100 picoseconds. If you do not specify this keyword, the default is 1 ns. You can specify the following time specifications: ns (nanoseconds; default) us (microseconds) ps (picoseconds)
CELL	<i>cell_constructs</i>	See " <a href="#">CELL Keyword and Constructs</a> " on <a href="#">page 1003</a> for more information.

## CELL Keyword and Constructs

The cell entries identify specific design instances, paths, and nets and associate timing data with them. Cell entries are specific to a design, instance, library, or type. Each cell entry begins with the `CELL` keyword followed by the `CELLTYPE`, and `INSTANCE` keywords. These keywords, in turn, are followed by one or more timing specifications, which contain the actual timing data associated with the cell entry. The timing data is specified with the `INCLUDE`, `DELAY`, `TIMINGCHECK`, and `TIMINGENV` keywords.

The `CELL` keyword specifies an instance of a cell. The syntax for the `CELL` keyword is as follows:

```
(CELL
  (CELLTYPE "celltype")
  (INSTANCE path)
  {(INCLUDE path)}
  {(DELAY delay_keywords)}
  {(TIMINGCHECK tcheck_keywords)}
  {(TIMINGENV tenv_keywords)}
)
```

**Note:** The `TIMINGENV` keyword and its constructs are ignored by NC-Verilog, but are included in this syntax diagram for completeness. You do not receive error messages if you specify `TIMINGENV` keywords.

## **CELLTYPE Keyword**

The **CELLTYPE** keyword specifies the type of a cell. This keyword is equivalent to the HDL module name. Enclose this string in quotation marks. For example,

```
(CELL
  (CELLTYPE "DFF")
  ...
)
```

## **INSTANCE Keyword**

The **INSTANCE** keyword specifies an instance of the specified cell type. Specify a full hierarchical path such as `a1.b1.c1` or a path relative to the scope of annotation. A full hierarchical path can be represented in either of the following formats:

```
(CELL
  (CELLTYPE "DFF")
  (INSTANCE a1.b1.c1)
    timing_specification
    timing_specification
)
(CELL
  (CELLTYPE "DFF")
  (INSTANCE a1)
  (INSTANCE b1)
  (INSTANCE c1)
    timing_specification
    timing_specification
)
```

The timing data in the timing specification applies only to the specified cell instance. You can also specify an arrayed instance for the cell instance, as shown in the following format:

```
(CELL (CELLTYPE "DFF")
  (INSTANCE a1.b1[3].c1)
    timing_specification
    timing_specification
)
```

To associate the timing data with all instances of the specified cell type, place a wildcard character (\*) after the **INSTANCE** keyword, as shown in the following example. Only instances in or below the current top level are affected.

## NC-Verilog Simulator Help

### SDF File Syntax

---

```
(CELL (CELLTYPE "dff")
  (INSTANCE *)
    timing_specification
    timing_specification
    ...
)
```

If you do not specify a path, the default is the current top level.

## INCLUDE Keyword

The `INCLUDE` keyword specifies the full or relative path of a file that contains timing specifications. The file is read as if it was inserted as a continuation of the current SDF file. If the specified file is in your current directory, you can specify just the file name.

The following example shows how to use the `INCLUDE` keyword.

```
(CELL (CELLTYPE "dff")
  (INSTANCE a.b.c)
  (INCLUDE /cds/home/dff_celldef)
)
```

## DELAY Keyword and Constructs

The **DELAY** keyword specifies the delay values associated with module paths, nets, interconnects, devices, and ports. You can specify the keyword entries listed in the following syntax.

```
(DELAY
  { (ABSOLUTE | INCREMENT
    (IOPATH port_spec port_path delay_list)
    (COND cond_port_expr
      (IOPATH port_spec port_path
        { (RETAIN delay_list) } delay_list))
    (CONDELS cond_port_expr
      (IOPATH port_spec port_path
        { (RETAIN delay_list) } delay_list))
    (PORT port_path delay_list)
    (INTERCONNECT port_path1 port_path2 delay_list)
    (NETDELAY name delay_list)
    (DEVICE {port_path} delay_list))
  { (PATHPULSE port_path1 {port_path2} (reject) {(error)}})
  { (PATHPULSEPERCENT port_path1 {port_path2} (reject) {(error)} )}
)
)
```

The *delay\_list* variable can specify one, two, three, six, or twelve delays. A delay can be a single value or three values representing minimum, typical, and maximum delays in the form *min:typ:max*.

The following table shows the transitions for which you can specify delays:

Transitions	Delay Syntax
All transitions	<i>(delay)</i>
Rise and fall	<i>(delay) (delay)</i>
Rise, fall, and Z	<i>(delay) (delay) (delay)</i>
01, 10, 0Z, Z1, 1Z, Z0	<i>(delay) (delay) (delay) (delay) (delay) (delay)</i>
01, 10, 0Z, Z1, 1Z, Z0, 0X, X1, 1X, X0, XZ, ZX	<i>(delay) (delay) (delay) (delay) (delay) (delay) (delay) (delay) (delay) (delay) (delay) (delay)</i>

## **ABSOLUTE Keyword**

The ABSOLUTE keyword specifies that the delay values replace the existing delay values in the design. You can use positive or negative numbers with the ABSOLUTE keyword. The syntax is as follows:

```
(ABSOLUTE
  (IOPATH port_spec port_path delay_list)
  (COND cond_port_expr
    (IOPATH port_spec port_path
      { (RETAIN delay_list) } delay_list))
  (CONDELSE cond_port_expr
    (IOPATH port_spec port_path
      { (RETAIN delay_list) } delay_list))
  (PORT port_path delay_list)
  (INTERCONNECT port_path1 port_path2 delay_list)
  (NETDELAY name delay_list)
  (DEVICE {port_path} delay_list)
)
)
```

In the following example, the delay values are specified as two *min:typ:max* triplets. The first triplet is the delay for a rising edge transition and the second triplet is the delay for a falling edge transition.

```
(CELL
  (CELLTYPE "DFF")
  (INSTANCE a.b.c)
  (DELAY
    (ABSOLUTE
      (IOPATH (posedge clk) q (22:28:33) (25:30:37))
      (PORT clr (32:39:49) (35:41:47))
    )
  )
)
```

## INCREMENT Keyword

The **INCREMENT** keyword specifies that the delay values are added to the existing delay values. You can use positive or negative numbers with the **INCREMENT** keyword. The syntax is as follows:

```
(INCREMENT
  (IOPATH port_spec port_path delay_list)
  (COND cond_port_expr
    (IOPATH port_spec port_path
      { (RETAIN delay_list) } delay_list))
  (CONDELSE cond_port_expr
    (IOPATH port_spec port_path
      { (RETAIN delay_list) } delay_list))
  (PORT port_path delay_list)
  (INTERCONNECT port_path1 port_path2 delay_list)
  (NETDELAY name delay_list)
  (DEVICE {port_path} delay_list)
)
)
```

In the following example, the delay values are specified as two *min:typ:max* triplets. The first triplet is the delay for a rising edge transition and the second triplet is the delay for a falling edge transition.

```
(CELL (CELLTYPE "DFF")
  (INSTANCE a.b.c)
  (DELAY (INCREMENT
    (IOPATH (posedge clk) q (-4::2) (-7::5))
    (PORT clr (2:3:4) (5:6:7))
  )
)
)
```

## IOPATH Keyword

The **IOPATH** keyword specifies delays on a path from an input port to an output port of a device and optional reject limits and error limits on the path. The syntax for the **IOPATH** keyword is as follows:

```
(IOPATH port_spec port_path {(RETAIN delay_list)} delay_list)
```

---

<b>Keyword Argument</b>	<b>Description</b>
<i>port_spec</i>	Input or inout (bidirectional) port. An edge identifier can be included.
<i>port_path</i>	Output or inout port. Where applicable, a port path can have an array index (for example, <i>x</i> . <i>y</i> [3]. <i>p</i> ).
RETAIN	Ignored by NC-Verilog. It is provided in the syntax for completeness.
<i>delay_list</i>	IOPATH delay from <i>port_spec</i> to <i>port_path</i> . The delay can also include optional reject and error limit specifications. You can specify negative numbers only within the INCREMENT keyword construct.

---

Each delay value is associated with a unique input port/output port pair.

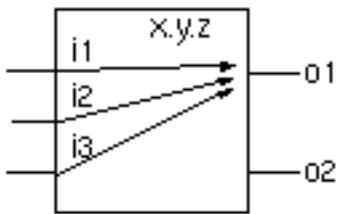
Use the **COND** keyword to specify conditional module path delays.

In the following example, the *delay\_list* for each **IOPATH** is a set of three *min:typ:max* triplets that specifies the delays for rise, fall, and turn-off transitions. This example also includes a conditional IOPATH using the **COND** construct to represent state-dependent path delays.

## NC-Verilog Simulator Help

### SDF File Syntax

```
(INSTANCE x.y.z)
(DELAY (ABSOLUTE
        (IOPATH (posedge i1) o1 (2:3:4) (4:5:6) (3:5:6))
        (IOPATH i2 o1 (2:4:5) (5:6:7) (4:6:7))
        (COND i1 (IOPATH i3 o1 (2:4:5) (4:5:6) (4:5:7)))
    )
)
```



To specify optional reject and error limits, enclose the entire *delay\_list* in parentheses and enclose the delay, reject limit, and error limit in their own parentheses. For example, the following construct specifies one delay. For all transitions, the delay is 12, the reject limit is 6, and the error limit is 10.

```
(IOPATH A B ((12:12:12) (6:6:6) (10:10:10)))
```

**Note:** The SDF Annotator calculates the reject and error limit values as follows to determine the level of acceptance for a delay value on a module path.

```
error_limit = (error%/100) * (module_path_delay)  
reject_limit = (reject%/100) * (module_path_delay)
```

To specify that a current delay is to be maintained, use an empty set of parentheses. For example, the following IOPATH statement annotates a delay of 3 : 5 : 7 and an error limit of 2 : 3 : 6, while keeping the current setting for the reject limit.

```
(IOPATH A B ((3:5:7) ( ) (2:3:6)))
```

The following commented examples illustrate the syntax for the IOPATH keyword construct.

```
(IOPATH A B (12:12:12))
    // Delay=12 for all transitions.
    // Reject and error limits are not specified, and are set equal to the delay.

(IOPATH A B (12:12:12)
    (10:10:10))
    // Delay=12 for rise transition. Delay=10 for fall transition.
    // Reject and error limits are not specified, and are set equal to the delay.

(IOPATH A B ((12:12:12) (10:10:10)))
    // Delay=12 and reject=10 for all transitions.
```

## NC-Verilog Simulator Help

### SDF File Syntax

---

```
// Error limit is not included, so it is set equal to reject limit.

(IOPATH A B ((12:12:12) (6:6:6) (10:10:10)))
    // Delay=12, reject=6, error=10 for all transitions.

(IOPATH A B ((12:12:12) ( ) (10:10:10)))
    // Delay=12, reject=current value, error=10 for all transitions.

(IOPATH A B (12:12:12)
    ((10:10:10) (5:5:5) (9:9:9)))
    // Delay=12, reject=12, error=12 for rise transition.
    // Delay=10, reject=5, error=9 for fall transition.

(IOPATH A B ((12:12:12) (6:6:6) (8:8:8))
    ((10:10:10) (5:5:5) (9:9:9)))
    // Delay=12, reject=6, error=8 for rise transition.
    // Delay=10, reject=5, error=9 for fall transition.

(IOPATH A B ((12:12:12) (6:6:6))
    ((10:10:10) (5:5:5) (9:9:9)))
    // Delay=12, reject=6, error=6 for rise transition.
    // Delay=10, reject=5, error=9 for fall transition.

(IOPATH A B ((5:5:5) (2:2:2) (3:3:3))
    ((6:6:6) (3:3:3) (4:4:4))
    ((15:15:15) (7:7:7) (10))
        ((14:14:14) (6:6:6) (9:9:9))
    ((12:12:12) (7:7:7) (9:9:9))
        ((13:13:13) (5:5:5) (8:8:8)))
    // Delay=5, reject=2, error=3 for 01 transition.
    // Delay=6, reject=3, error=4 for 10 transition.
    // Delay=15, reject=7, error=10 for 0Z transition.
    // Delay=14, reject=6, error=9 for Z1 transition.
    // Delay=12, reject=7, error=9 for 1Z transition.
    // Delay=13, reject=5, error=8 for Z0 transition.
```

## COND Keyword

The COND keyword specifies conditional module path delays. The syntax is as follows:

```
(COND cond_port_expr
  (IOPATH port_spec port_path
    {(RETAIN delay_list)} delay_list
  )
)
```

Keyword Argument	Description
<i>cond_port_expr</i>	Boolean description of the state dependency of the delay. The delay values apply if the <i>cond_port_expr</i> is true (logical one) and do not apply if the <i>cond_port_expr</i> is false (logical 0).
IOPATH	See “ <a href="#">IOPATH Keyword</a> ” on page 1009 for more information.
<i>port_spec</i>	Input or inout port that can have an edge identifier.
<i>port_path</i>	Output or inout port. Where applicable, a port path can have array index (for example, <i>x.y[3].p</i> ).
RETAIN	Ignored by NC-Verilog. It is provided in the syntax for completeness.
<i>delay_list</i>	IOPATH delay from <i>port_spec</i> to <i>port_path</i> .

The following example shows a conditional IOPATH using the COND construct to represent state-dependent path delays.

```
(INSTANCE x.y.z)
(DELAY (ABSOLUTE
  (COND i1 (IOPATH i3 o1 (2:4:5) (4:5:6) (4:5:7))
  )
)
```

## **CONDELSE Keyword**

The CONDELSE keyword specifies path delays when a signal change propagates to an output or inout, but none of the conditions for module paths to it are true. The syntax is as follows:

```
(CONDELSE (IOPATH port_spec port_path delay_list) )
```

See the “[COND Keyword](#)” on page 1012 section for details.

Use the CONDELSE keyword to cover all cases for a path that have not been specified in COND keyword constructs.

## **RETAIN Keyword**

**Note:** The RETAIN construct is ignored by NC-Verilog.

The RETAIN keyword specifies the time for which an output or inout port retains its previous logic value after a change at a related input or inout port. It is specified inside an IOPATH keyword construct. Use the RETAIN keyword on paths that proceed from the address or select inputs to the data outputs of memory and register file circuits. Use this keyword only where the cell timing model includes an explicit mechanism for providing retention times.

The syntax for RETAIN is as follows:

```
(RETAIN delay_list)
```

The *delay\_list* variable specifies the retention time data from the *port\_spec* to the *port\_path* variables in the IOPATH keyword construct. Consecutive delays in the *delay\_list* must be increasing numerically.

The following example shows the retain time of the bus `do[7:0]` with respect to changes on the bus `addr[13:0]`. The rise time (`4:5:7`) proceeds from low to X; the fall time (`5:6:9`) proceeds from high to X.

```
(IOPATH addr[13:0] do[7:0] (RETAIN (4:5:7) (5:6:9))
```

## PORT Keyword

The **PORT** keyword specifies estimated or actual interconnect delay values you can place on the input port, without having to specify a start point for the wire path. The syntax is as follows:

(PORT *port\_path* *delay\_list*)

---

Keyword Argument	Description
<i>port_path</i>	Input or inout port. Where applicable, a port path can have array index (for example, <i>x.y[3].p</i> ).
<i>delay_list</i>	PORT delay of the <i>port_path</i> . Optional reject and error limits can be included.

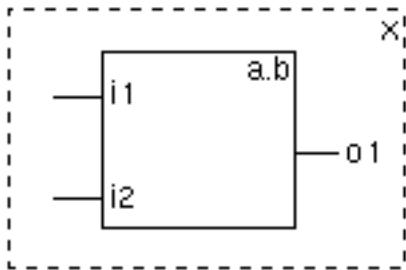
---

If you have multiple port delays to the same port, the maximum delay is selected by default.

If you specify interconnect delays with the **INTERCONNECT** keyword and port delays with the **PORT** keyword for the same input of a module, the annotator maps the interconnect delays to port delays and then selects the maximum delay by default.

In the following example, the delay consists of one *min:typ:max* triplet specifying the minimum, typical, and maximum delays for all transitions. The **PORT** delay is specified as incremental, which means the existing delay data values are increased or decreased rather than replaced.

```
(INSTANCE x)
(DELAY
  (INCREMENT (PORT a.b.i1 (-2:0:2))
  )
)
```



## NC-Verilog Simulator Help

### SDF File Syntax

---

By default, PORT delays are mapped to the default type of interconnect delay, which use inertial delays. Use the `-intermod_path` command-line option when you invoke the elaborator (`ncelab`) to enable transport delay functionality with full pulse control.

To specify optional reject and error limits, enclose the entire `delay_list` in parentheses and enclose the delay, reject limit, and error limit in their own parentheses. For example, the following command specifies one delay. For all transitions, the delay is 12, the reject limit is 6, and the error limit is 10.

```
(PORT A ((12:12:12) (6:6:6) (10:10:10)))
```

To specify that a current delay is to be maintained, use an empty set of parentheses. For example, the following PORT keyword annotates a delay of 3:5:7 and an error limit of 2:3:6, while keeping the current setting for the reject limit.

```
(PORT A ((3:5:7) ( ) (2:3:6)))
```

The following commented examples show how to use the PORT keyword.

```
(PORT A (12:12:12)
    // Delay=12 for all transitions
    // Reject and error limits are not specified, and are set equal to the delay.

(PORT A (12:12:12)
    (10:10:10))
    // Delay=12 for rise transition. Delay=10 for fall transition.
    // Reject and error limits are not specified, and are set equal to the delay.

(PORT A ((12:12:12) (6:6:6) (10:10:10)))
    // Delay=12, reject=6, error=10 for all transitions.

(PORT A ((12:12:12) ( ) (10:10:10)))
    // Delay=12, reject=current value, error=10 for all transitions.

(PORT A ((12:12:12) (10:10:10)))
    // Delay=12 and reject=10 for all transitions.
    // Error limit is not included, so it is set equal to reject limit.

(PORT A ((5:5:5) (2:2:2) (3:3:3))
    ((6:6:6) (3:3:3) (4:4:4))
    ((15:15:15) (7:7:7) (10))
    ((14:14:14) (6:6:6) (9:9:9))
    ((12:12:12) (7:7:7) (9:9:9))
    ((13:13:13) (5:5:5) (8:8:8)))
    // Delay=5, reject=2, error=3 for 01 transition.
    // Delay=6, reject=3, error=4 for 10 transition.
```

## NC-Verilog Simulator Help

### SDF File Syntax

---

```
// Delay=15, reject=7, error=10 for 0Z transition.  
// Delay=14, reject=6, error=9 for Z1 transition.  
// Delay=12, reject=7, error=9 for 1Z transition.  
// Delay=13, reject=5, error=8 for Z0 transition.
```

## INTERCONNECT Keyword

The INTERCONNECT keyword specifies estimated or actual delays in the wire paths between devices. The syntax is as follows:

```
(INTERCONNECT port_path1 port_path2 delay_list)
```

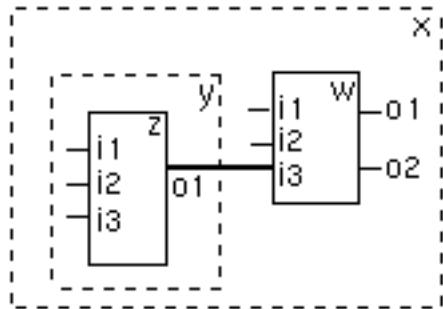
---

Keyword Argument	Description
<i>port_path1</i>	Output or inout port. Where applicable, a port path can have array index (for example, x.y[3].p).
<i>port_path2</i>	Input or inout port. Where applicable, a port path can have array index (for example, x.y[3].p).
<i>delay_list</i>	Interconnect delay between the output and input ports. Unique delays can be specified for multi-source nets. The delay can also include optional reject and error limits.

---

In the following example, the *delay\_list* consists of two *min:typ:max* triplets specifying the delays for rise and fall transitions.

```
(INSTANCE x)  
(DELAY  
  (ABSOLUTE  
    (INTERCONNECT y.z.o1 w.i3 (5:6:7) (5.5:6:6.5))  
  )  
)
```



If you specify interconnect delays with the `INTERCONNECT` keyword and port delays with the `PART` keyword for the same input of a module, the annotator maps the interconnect delays to port delays and then selects the maximum delay by default.

By default, `INTERCONNECT` delays are mapped to the default type of interconnect delay, which use inertial delays. Use the `-intermod_path` command-line option when you invoke the elaborator (`ncelab`) to enable transport delay functionality with full pulse control.

To specify optional reject and error limits, enclose the entire `delay_list` in parentheses and enclose the delay, reject limit, and error limit in their own parentheses. For example, the following command specifies one delay. For all transitions, the delay is 12, the reject limit is 6, and the error limit is 10.

```
(INTERCONNECT A B ((12:12:12) (6:6:6) (10:10:10)))
```

To specify that a current delay is to be maintained, use an empty set of parentheses. For example, the following `INTERCONNECT` statement annotates a delay of 3:5:7 and an error limit of 2:3:6, while keeping the current setting for the reject limit.

```
(INTERCONNECT A B ((3:5:7) ( ) (2:3:6)))
```

The following commented examples illustrate the syntax for `INTERCONNECT`.

```
(INTERCONNECT A B (12:12:12))
  // Delay=12 for all transitions.
  // Reject and error limits are not specified, and are set equal to the delay.

(INTERCONNECT A B ((12:12:12) (6:6:6) (10:10:10)))
  // Delay=12, reject=6, error=10 for all transitions.

(INTERCONNECT A B ((12:12:12) ( ) (10:10:10)))
  // Delay=12, reject=current value, error=10 for all transitions.

(INTERCONNECT A B ((12:12:12) (10:10:10)))
  // Delay=12 and reject=10 for all transitions.
  // Error limit is not included, so it is set equal to reject limit.
```

## NC-Verilog Simulator Help

### SDF File Syntax

---

```
(INTERCONNECT A B (12:12:12)
    ((10:10:10) (5:5:5) (9:9:9)))
// Delay=12, reject=12, error=12 for rise transition.
// Delay=10, reject=5, error=9 for fall transition.

(INTERCONNECT A B ((5:5:5) (2:2:2) (3:3:3))
    ((6:6:6) (3:3:3) (4:4:4))
    ((15:15:15) (7:7:7) (10))
    ((14:14:14) (6:6:6) (9:9:9))
    ((12:12:12) (7:7:7) (9:9:9))
    ((13:13:13) (5:5:5) (8:8:8)))
// Delay=5, reject=2, error=3 for 01 transition.
// Delay=6, reject=3, error=4 for 10 transition.
// Delay=15, reject=7, error=10 for 0Z transition.
// Delay=14, reject=6, error=9 for Z1 transition.
// Delay=12, reject=7, error=9 for 1Z transition.
// Delay=13, reject=5, error=8 for Z0 transition.

(INTERCONNECT A D ((5:5:5) (2:2:2) (3:3:3)))
(INTERCONNECT B D ((6:6:6) (3:3:3) (4:4:4)))
(INTERCONNECT C D ((7:7:7) (4:4:4) (5:5:5)))
// Unique delays, reject limits, and error limits for multi-source net.
```

## NETDELAY Keyword

**Note:** This construct is not supported in the OVI SDF Version 3 specification. Specify a version number that is lower than 3.0 with the SDFVERSION statement (for example, SDFVERSION "2.1") to use the NETDELAY construct.

The NETDELAY keyword specifies delay for a complete net, where delays from all the source port(s) on the net to all destination port(s) have the same value. NETDELAY is a short form of INTERCONNECT delay.

The syntax is as follows:

```
(NETDELAY name delay_list)
```

## NC-Verilog Simulator Help

### SDF File Syntax

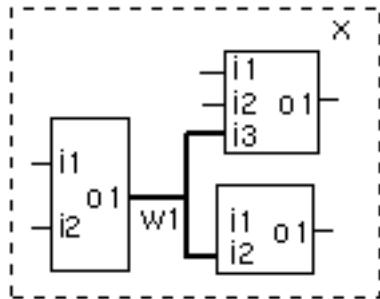
---

Keyword Argument	Description
<i>name</i>	Name of the net or the output port driving the net. Where applicable, a port name can have array index (for example, <i>x.y[3].p</i> ).
<i>delay_list</i>	Delay associated with the net or port specified by name. The value specifies the same delay for all source/load pairs. The delay can include optional reject and error limits.

---

In the following example, the net is identified by name. The *delay\_list* consists of three *min : typ : max* triplets specifying the rise, fall, and turn-off delays.

```
(INSTANCE x)
(DELAY
  (ABSOLUTE
    (NETDELAY w1 (2.5:3.0:3.5) (2.9:3.4:4.2) (6.3:8:9.9)) ))
```



By default, NETDELAY delays are mapped to the default type of interconnect delay, which use inertial delays. Use the `-intermod_path` command-line option when you invoke the elaborator (*ncelab*) to enable transport delay functionality with full pulse control.

To specify optional reject and error limits, enclose the entire *delay\_list* in parentheses and enclose the delay, reject limit, and error limit in their own parentheses. For example, the following command specifies one delay. For all transitions, the delay is 12, the reject limit is 6, and the error limit is 10.

```
(NETDELAY A ((12:12:12) (6:6:6) (10:10:10)))
```

To specify that a current delay is to be maintained, use an empty set of parentheses. For example, the following NETDELAY statement annotates a delay of 3 : 5 : 7 and an error limit of 2 : 3 : 6, while keeping the current setting for the reject limit.

## NC-Verilog Simulator Help

### SDF File Syntax

---

```
(NETDELAY A ((3:5:7) ( ) (2:3:6)))
```

The following commented examples illustrate the syntax for NETDELAY.

```
(NETDELAY A (12:12:12))
```

// Delay=12 for all transitions.

// Reject and error limits are not specified, and are set equal to the delay.

```
(NETDELAY A (12:12:12)
```

```
    (10:10:10))
```

// Delay=12 for rise transition. Delay=10 for fall transition.

// Reject and error limits are not specified, and are set equal to the delay.

```
(NETDELAY A ((12:12:12) (6:6:6) (10:10:10)))
```

// Delay=12, reject=6, error=10 for all transitions.

```
(NETDELAY A ((12:12:12) ( ) (10:10:10)))
```

// Delay=12, reject=current value, error=10 for all transitions.

```
(NETDELAY A ((12:12:12) (10:10:10)))
```

// Delay=12 and reject=10 for all transitions.

// Error limit is not included, so it is set equal to reject limit.

```
(NETDELAY A ((5:5:5) (2:2:2) (3:3:3))
```

```
    ((6:6:6) (3:3:3) (4:4:4))
```

```
    ((15:15:15) (7:7:7) (10))
```

```
    ((14:14:14) (6:6:6) (9:9:9))
```

```
    ((12:12:12) (7:7:7) (9:9:9))
```

```
    ((13:13:13) (5:5:5) (8:8:8)))
```

// Delay=5, reject=2, error=3 for 01 transition.

// Delay=6, reject=3, error=4 for 10 transition.

// Delay=15, reject=7, error=10 for 0Z transition.

// Delay=14, reject=6, error=9 for Z1 transition.

// Delay=12, reject=7, error=9 for 1Z transition.

// Delay=13, reject=5, error=8 for Z0 transition.

## DEVICE Keyword

The **DEVICE** keyword specifies the intrinsic delay of a module or gate. Intrinsic delay is specific to the type of the object and has the same value for every instance of that module or gate. Conceptually, this represents all path delays through the object, independent of loading or input slope. At the gate level, the delay is associated with the output. If a module has more than one output, specify the delays to each output port by using additional *port\_path* specifications. If you do not specify any port, SDF assumes that all output ports have the same delay values. The syntax is as follows:

```
(DEVICE {port_path} delay_list)
```

---

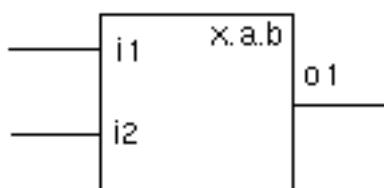
Keyword Argument	Description
<i>port_path</i>	Gate primitive instance, module instance, or output or inout port. Where applicable, a port path can have an array index (for example, <i>x.y[ 3 ].p</i> ).
<i>delay_list</i>	Device delay (specific to a type). The delay can include optional reject and error limits.

---

- If *port\_path* is a gate primitive, delays are annotated to the gate primitive.
- If *port\_path* is a module instance, delays are annotated to all paths to all outputs.
- If *port\_path* is an output or inout port, delays are annotated to all paths to that output or inout.

In the following example, the *delay\_list* consists of three *min:typ:max* triplets specifying the rise, fall, and turn-off delays.

```
(INSTANCE x.a.b)
(DELAY
  (ABSOLUTE
    (DEVICE o1 (6:7:8) (4:6:7) (5:8:9))
  )
)
```



## NC-Verilog Simulator Help

### SDF File Syntax

---

To specify optional reject and error limits, enclose the entire *delay\_list* in parentheses and enclose the delay, reject limit, and error limit in their own parentheses. For example, the following command specifies one delay. For all transitions, the delay is 12, the reject limit is 6, and the error limit is 10.

```
(DEVICE A ((12:12:12) (6:6:6) (10:10:10)))
```

To specify that you want a current delay maintained, use an empty set of parentheses. For example, the following DEVICE statement annotates a delay of 3 : 5 : 7 and an error limit of 2 : 3 : 6, while keeping the current setting for the reject limit.

```
(DEVICE A ((3:5:7) ( ) (2:3:6)))
```

The following commented examples illustrate the syntax for DEVICE.

```
(DEVICE A (12:12:12)
    // Delay=12 for all transitions.
    // Reject and error limits are not specified, and are set equal to the delay.

(DEVICE A (12:12:12)
    (10:10:10))
    // Delay=12 for rise transition. Delay=10 for fall transition.
    // Reject and error limits are not specified, and are set equal to the delay.

(DEVICE A ((12:12:12) (10:10:10)))
    // Delay=12 and reject=10 for all transitions.
    // Error limit is not included, so it is set equal to reject limit.

(DEVICE A ((12:12:12) (6:6:6) (10:10:10)))
    // Delay=12, reject=6, error=10 for all transitions.

(DEVICE A ((12:12:12) ( ) (10:10:10)))
    // Delay=12, reject=current value, error=10 for all transitions.

(DEVICE A (12:12:12)
    ((10:10:10) (5:5:5) (9:9:9)))
    // Delay=12, reject=12, error=12 for rise transition.
    // Delay=10, reject=5, error=9 for fall transition.

(DEVICE A ((12:12:12) (6:6:6) (8:8:8))
    ((10:10:10) (5:5:5) (9:9:9)))
    // Delay=12, reject=6, error=8 for rise transition.
    // Delay=10, reject=5, error=9 for fall transition.
```

```
(DEVICE A ((5:5:5) (2:2:2) (3:3:3))
          ((6:6:6) (3:3:3) (4:4:4))
          ((15:15:15) (7:7:7) (10))
          ((14:14:14) (6:6:6) (9:9:9))
          ((12:12:12) (7:7:7) (9:9:9))
          ((13:13:13) (5:5:5) (8:8:8)))
// Delay=5, reject=2, error=3 for 01 transition.
// Delay=6, reject=3, error=4 for 10 transition.
// Delay=15, reject=7, error=10 for 0Z transition.
// Delay=14, reject=6, error=9 for Z1 transition.
// Delay=12, reject=7, error=9 for 1Z transition.
// Delay=13, reject=5, error=8 for Z0 transition.
```

## PATHPULSE Keyword

The **PATHPULSE** keyword specifies the limits associated with a legal path between an input port and an output port of a device. These limits determine whether a pulse at the input can pass through the device to the output.

The syntax for the **PATHPULSE** keyword is as follows. If you specify only one value for *reject* or *error*, both limits are set to that value.

```
(PATHPULSE port_path1 {port_path2} (reject) {(error)} )
```

---

<b>Keyword Argument</b>	<b>Description</b>
<i>port_path1</i>	Input or inout port. Where applicable, a port path can have array index (for example, <i>x.y[3].p</i> ).
<i>port_path2</i>	Output or inout port. Where applicable, a port path can have array index (for example, <i>x.y[3].p</i> ).
<i>reject</i>	Pulse rejection limit, in time units. This limit defines the minimum pulse width required for the pulse to pass through to the output. Anything smaller does not affect the output.
<i>error</i>	The error limit, in time units. This limit defines the minimum pulse width necessary to drive the output to a known state. Anything smaller causes the output to enter the unknown (e) state or is rejected (if smaller than the pulse reject limit). The error limit must be equal to or greater than the pulse reject limit.

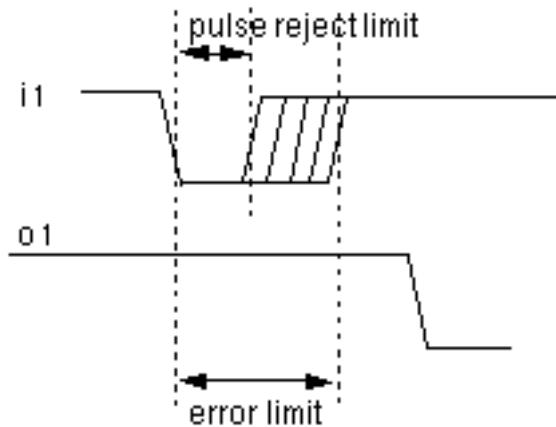
---

## NC-Verilog Simulator Help

### SDF File Syntax

In the following example, the first value (13) is the pulse reject limit and the second value (21) is the error limit.

```
(INSTANCE x)
(DELAY
  (PATHPULSE y.i1 y.o1 (13) (21))
)
```



## PATHPULSEPERCENT Keyword

The PATHPULSEPERCENT keyword is the same as the PATHPULSE keyword except that reject and error limits are expressed in percentages (%) of the cell path delay from the input to the output. If you specify only one value, both reject and error limits are set to that value. See the [PATHPULSE](#) keyword for a description of the syntax.

**Note:** The PATHPULSEPERCENT keyword supersedes the GLOBALPATHPULSE keyword.

In the following example, the first value (25) is the pulse reject limit and the second value (35) is the error limit.

```
(INSTANCE x)
(DELAY
  (PATHPULSEPERCENT y.i1 y.o1 (25) (35))
)
```

The error limit must be equal to or greater than the reject limit.

If you omit both the reject limit and error limit, both specifications are set to 100. If the reject limit exceeds the error limit or if you omit the error limit, the error limit is set equal to the reject limit.

## TIMINGCHECK Keyword and Constructs

The TIMINGCHECK keyword assigns timing check limits to specific cell instances and assigns layout constraints to critical paths in the design that determine how the signals can change in relation to each other. EDA tools use this information during the design process, as follows:

- Simulation tools are notified of signal transitions that violate timing checks.
- Timing analysis tools identify paths that might violate timing checks and determine the timing constraints for those paths.
- Layout tools use the timing constraints from timing analysis tools to generate layouts that do not violate any timing checks.

The syntax of the TIMINGCHECK keyword is as follows:

```
(TIMINGCHECK
  {(SETUP data_sig clk_sig setup_time)}

  {(HOLD data_event clk_event hold_time)}

  {(SETUPHOLD data_event clk_event setup_time hold_time
    {SCOND tstamp_cond} {CCOND tcheck_cond} )}

  {(RECOVERY asynch_ctl_sig clk_or_gate recovery_time
    {SCOND tstamp_cond} {CCOND tcheck_cond} )}

  {(REMOVAL asynch_ctl_sig clk_or_gate removal_time
    {SCOND tstamp_cond} {CCOND tcheck_cond} )}

  {(RECREM asynch_ctl_port clk_or_gate recovery_time removal_time
    {SCOND tstamp_cond} {CCOND tcheck_cond} )}

  {(SKEW lower_clk upper_clk max_skew)}

  {(WIDTH edge_clk max_width)}

  {(PERIOD edge_clk max_period)}

  {(NOCHANGE clk_event data_event start_offset end_offset)}
)
```

## NC-Verilog Simulator Help

### SDF File Syntax

---

**Note:** The following syntax applies to SDF standards prior to OVI Version 2.0, where *tcheck* applies to the TIMINGCHECK keyword constructs.

```
(TIMINGCHECK  
  (COND tcheck_cond tcheck)  
)
```

The following example shows a cell entry that assigns setup and hold timing checks:

```
(CELL (CELLTYPE "DFF")  
  (INSTANCE a.b.c)  
  (TIMINGCHECK  
    (SETUP din (posedge clk) (3:4:5.5))  
    (HOLD din (posedge clk) (4:5.5:7))  
  )  
)
```

## COND Keyword

The COND keyword specifies conditional timing check. The syntax is as follows for OVI SDF Versions previous to 2.0:

```
(COND tcheck_cond tcheck)
```

The COND keyword in OVI SDF Version 2.0 and higher places the condition on the signal itself as shown in the following syntax examples:

```
(SETUP (COND tcheck_cond data_sig) clk_sig setup_time)  
(SETUP (data_sig (COND tcheck_cond clk_sig) setup_time)
```

---

Keyword Argument	Description
<i>tcheck_cond</i>	Boolean expression.
<i>data_sig</i>	Data signal.

---

**Note:** In SETUPHOLD and RECOVERY timing checks, you can use the SCOND and CCOND constructs to condition the time stamp and time check events. See “[SETUPHOLD Keyword](#)” on page 1029 and “[RECOVERY Keyword](#)” on page 1030 for more information about these conditional keywords.

## SETUP Keyword

The SETUP keyword specifies the minimum interval before a clock transition. The syntax is as follows:

```
(SETUP data_sig clk_sig setup_time)
```

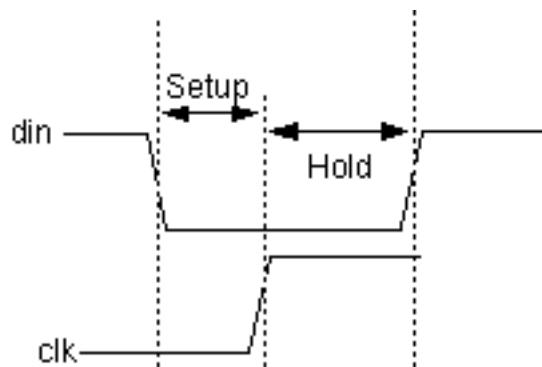
---

Keyword Argument	Description
<i>data_sig</i>	Data signal.
<i>clk_sig</i>	Clock signal.
<i>setup_time</i>	Specifies the minimum interval between the data and clock event (that is, <i>before</i> the clock transition). Any change to the data signal within these intervals results in a timing violation.

---

The following example shows a SETUP timing check:

```
(INSTANCE x.a)
(TIMINGCHECK
  (SETUP din (posedge clk) (12) ))
```



## HOLD Keyword

The `HOLD` keyword specifies the minimum interval after a clock transition. The syntax is as follows:

```
(HOLD data_event clk_event hold_time)
```

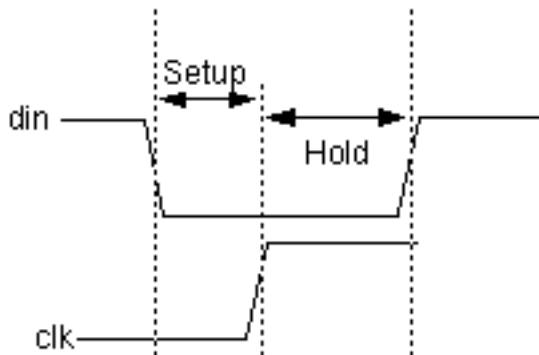
---

Keyword Argument	Description
<i>data_event</i>	Data event.
<i>clk_event</i>	Clock event.
<i>hold_time</i>	Specifies the minimum interval between the clock and data events (that is, <i>after</i> clock transition). Any change to the data signal within these intervals results in a timing violation.

---

The following example shows a `HOLD` timing check:

```
(INSTANCE x.a)
(TIMINGCHECK
  (HOLD din (posedge clk) (9.5) ) )
```



## SETUPHOLD Keyword

The SETUPHOLD keyword specifies the same information with one keyword as both the SETUP and HOLD keywords do. The syntax is as follows:

```
(SETUPHOLD data_event clk_event setup_time hold_time
           {(SCOND tstamp_cond)} {(CCOND tcheck_cond)} )
```

Keyword Argument	Description
<i>data_event</i>	Data event.
<i>clk_event</i>	Clock event.
<i>setup_time</i>	Minimum interval between the data and clock events (that is, <i>before</i> clock transition). Any change to the data signal within these intervals results in a timing violation.
<i>hold_time</i>	Minimum interval between the clock and data events (that is, <i>after</i> clock transition). Any change to the data signal within these intervals results in a timing violation.
(SCOND <i>tstamp_cond</i> )	Event that triggers the timing check. If the <i>tstamp_cond</i> is true, the timing check is accepted. If false, it is ignored.
(CCOND <i>tcheck_cond</i> )	Event that causes the timing check to be validated. If the <i>tcheck_cond</i> is true, the timing check is accepted. If false, it is ignored.

**Note:** The *setup\_time* or the *hold\_time* can be negative, but their sum must be 0 or more. To perform SDF annotation to SETUPHOLD timing checks with negative values, you must use the `-neg_tchk` option on the command line when you invoke the elaborator. Beginning with the LDV 3.3 release, this option is the default.

In addition to performing the SETUP and HOLD operations, the SETUPHOLD keyword can have different conditions specified for the event that triggers the check and the event that causes the check to be validated.

If SCOND or CCOND is used with the COND construct, the COND construct is overruled.

The following examples show how to use the SETUPHOLD keyword:

```
(INSTANCE x.a)
(TIMINGCHECK
  (SETUPHOLD din (posedge clk) (12) (9.5))
```

```
)  
)  
(INSTANCE x.a)  
(TIMINGCHECK  
  (SETUPHOLD din (posedge clk) (12) (9.5) (SCOND !rst)  
   )  
)  
(INSTANCE x.a)  
(TIMINGCHECK  
  (SETUPHOLD din (posedge clk) (12) (9.5) (CCOND !rst)  
   )  
)
```

## RECOVERY Keyword

The RECOVERY keyword limits a change in an asynchronous control signal and the next clock pulse (for example, between clearbar and the clock for a flip-flop). If the clock signal violates the constraint, the output value is unknown. The syntax is as follows:

```
(RECOVERY asynch_ctl_sig clk_or_gate recovery_time  
  {(SCOND tstamp_cond)} {(CCOND tcheck_cond)}  
)
```

---

Keyword Argument	Description
<i>asynch_ctl_sig</i>	Asynchronous control signal, which normally has an edge identifier associated with it to indicate which transition corresponds to the release from the active state.
<i>clk_or_gate</i>	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.
<i>recovery_time</i>	Minimum interval between the release of the asynchronous control signal and the next active edge of the clock/gate event. The simulator uses the recovery limit for deterministic comparisons and does not admit x values.
(SCOND <i>tstamp_cond</i> )	Event that triggers the timing check. If the <i>tstamp_cond</i> is true, the timing check is accepted. If false, it is ignored.

## NC-Verilog Simulator Help

### SDF File Syntax

---

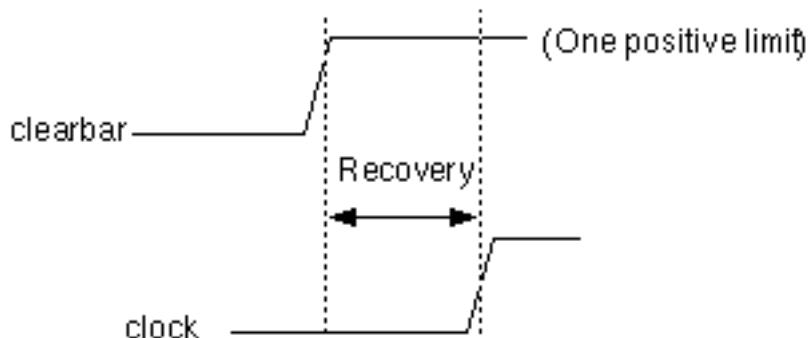
Keyword Argument	Description
(CCOND <i>tcheck_cond</i> )	Event that causes the timing check to be validated. If the <i>tcheck_cond</i> is true, the timing check is accepted. If false, it is ignored.

The recovery timing check can have different conditions specified for the event that triggers the check and the event that causes the check to be validated.

If SCOND or CCOND is used with the COND construct, the COND construct is overruled.

The following example shows how to use the RECOVERY keyword:

```
(INSTANCE x.b)
(TIMINGCHECK
  (RECOVERY (posedge clearbar) (posedge clk) (11.5)
  )
)
```



```
(INSTANCE x.a)
(TIMINGCHECK
  (RECOVERY (posedge rst) (posedge clk) (12) (9.5) (SCOND !clear)
  )
)
(INSTANCE x.a)
(TIMINGCHECK
  (RECOVERY (posedge rst) (posedge clk) (12) (9.5) (CCOND !clear)
  )
)
```

## REMOVAL Keyword

**Note:** NC-Verilog does not support the `REMOVAL` timing check.

The `REMOVAL` keyword is similar to the `HOLD` keyword. It specifies the time between an active clock edge and the release of an asynchronous control signal from the active state. The syntax is as follows:

```
(REMOVAL asynch_ctl_sig clk_or_gate removal_time
    { (SCOND tstamp_cond) } { (CCOND tcheck_cond) }
)
```

---

Keyword Argument	Description
<code>asynch_ctl_sig</code>	Asynchronous control signal, which normally has an edge identifier associated with it to indicate which transition corresponds to the release from the active state.
<code>clk_or_gate</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>removal_time</code>	Positive time value for which an extraordinary operation (such as a <code>set</code> or <code>reset</code> ) must persist to ensure that a device ignores any normal operation (such as, clocking in new data).
<code>( SCOND tstamp_cond )</code>	Event that triggers the timing check. If the <code>tstamp_cond</code> is true, the timing check is accepted. If false, it is ignored.
<code>( CCOND tcheck_cond )</code>	Event that causes the timing check to be validated. If the <code>tcheck_cond</code> is true, the timing check is accepted. If false, it is ignored.

---

For example, if the release of the clearbar occurs too soon after the edge of the clock, the state of a flip-flop between the clock and the clearbar becomes uncertain. That is, it could be the value set by the clearbar, or it could be the value clocked into the flip-flop from the data input. The following example shows how to use the `REMOVAL` keyword to avoid this problem.

```
(INSTANCE x.b)
(TIMINGCHECK
  (REMOVAL (posedge clearbar) (posedge clk) (6.3)
  )
)
```

## RECREM Keyword

The RECREM keyword specifies both recovery and removal limits in a single keyword. The syntax is as follows:

```
(RECREM asynch_ctl_sig clk_or_gate recovery_time removal_time
  {(SCOND tstamp_cond)} {(CCOND tcheck_cond)}
)
```

---

Keyword Argument	Description
<i>asynch_ctl_sig</i>	Asynchronous control signal, which normally has an edge identifier associated with it to indicate which transition corresponds to the release from the active state.
<i>clk_or_gate</i>	Clock (flip-flop) or gate (latch) signal, which normally has an edge identifier to indicate the active edge of the clock or the closing edge of the gate.
<i>recovery_time</i>	Minimum interval between the release of the asynchronous control signal and the next active edge of the clock/gate event. The simulator uses the recovery limit for deterministic comparisons and does not admit <i>x</i> values.
<i>removal_time</i>	Minimum interval between the clock and data events (that is, <i>after</i> clock transition). Any change to the data signal within these intervals results in a timing violation.
(SCOND <i>tstamp_cond</i> )	Event that triggers the timing check. If the <i>tstamp_cond</i> is true, the timing check is accepted. If false, it is ignored.
(CCOND <i>tcheck_cond</i> )	Event that causes the timing check to be validated. If the <i>tcheck_cond</i> is true, the timing check is accepted. If false, it is ignored.

---

When two time limits (*recovery\_time* and *removal\_time*) are specified, the *recovery\_time* can be negative, but the sum of both must be 0 or more. To perform SDF annotation to recovery timing checks with negative values, you must use the *-neg\_tchk* option on the command line when you invoke the elaborator. Beginning with the LDV 3.3 release, this option is the default.

The following example specifies a recovery time of 1.5 before the clock transition and a removal time of 0.8 after the clock transition. Any change to the clearbar signal within this interval results in a timing violation.

```
(INSTANCE x.b)
(TIMINGCHECK
  (RECREM (posedge clearbar) (posedge clk) (1.5) (0.8)
  )
)
```

## SKEW Keyword

The SKEW keyword specifies the limits for signal skew timing checks. A signal skew limit is the maximum delay allowable between two signals. You can specify these timing checks only between two signals existing at the same design hierarchy level. The syntax is as follows:

```
(SKEW lower_clk upper_clk max_skew)
```

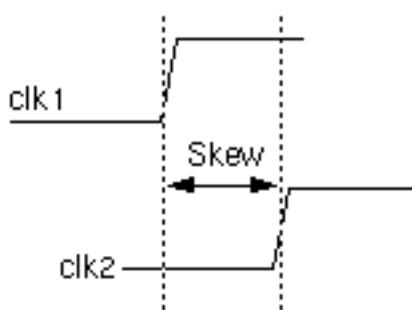
---

Keyword Argument	Description
<i>lower_clk</i>	Lower-bound clock event which can include an edge specification.
<i>upper_clk</i>	Upper-bound clock event which can include an edge specification.
<i>max_skew</i>	Maximum delay allowed between the upper- and lower-bound clock signals.

---

The following example shows how to use the SKEW keyword:

```
(INSTANCE x)
(TIMINGCHECK
  (SKEW (posedge clk1) (posedge clk2) (6) ) )
```



## **WIDTH Keyword**

The WIDTH keyword specifies the duration of signal levels from one clock edge to the opposite clock edge. If a signal has unequal phases, specify a separate width check for each phase. The syntax is as follows:

```
(WIDTH edge_clk max_width)
```

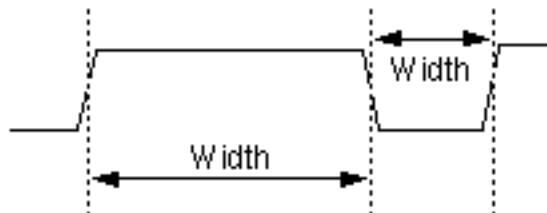
---

<b>Keyword Argument</b>	<b>Description</b>
<i>edge_clk</i>	Edge-triggered clock event.
<i>max_width</i>	Maximum time for the positive or negative phase of each cycle.

---

The following example shows how to use the WIDTH keyword:

```
(INSTANCE x.b)
(TIMINGCHECK
  (WIDTH (posedge clk) (30))
  (WIDTH (negedge clk) (16.5)))
)
```



The first width check is the phase beginning with the positive clock edge, and the second width check is the phase beginning with the negative clock edge. The data event is equal to the clock event with the opposite edge.

## PERIOD Keyword

The PERIOD keyword specifies limit values for a minimum period timing check. The minimum period timing check is the minimum allowable time for one complete cycle.

```
(PERIOD edge_clk max_period)
```

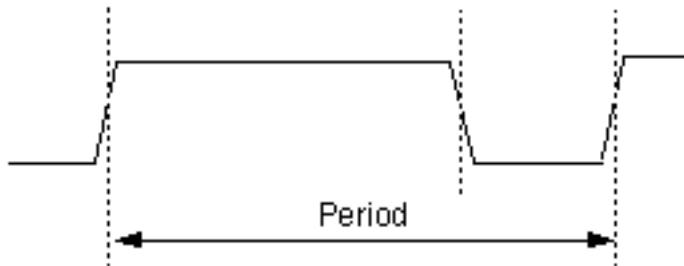
---

Keyword Argument	Description
<i>edge_clk</i>	Edge-triggered clock event.
<i>max_period</i>	Minimum period for complete signal cycle.

---

The following example shows how to use the PERIOD keyword. The period is the interval between two positive clock edges. The data event is equal to the clock event with the same edge.

```
(INSTANCE x.b)
(TIMINGCHECK
  (PERIOD (posedge clk) (46.5))
)
```



## NOCHANGE Keyword

**Note:** NC-Verilog does not support the NOCHANGE timing check.

The NOCHANGE keyword specifies a signal constraint relative to the width of a clock pulse. You can use this construct to model the timing constraints of memory devices, for example, when address lines must remain stable during a write pulse. The syntax is as follows:

```
(NOCHANGE clk_event data_event start_offset end_offset)
```

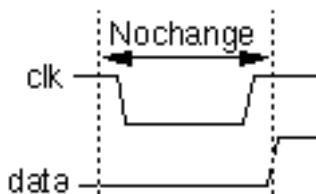
---

Keyword Argument	Description
<i>clk_event</i>	Clock event.
<i>data_event</i>	Data event.
<i>start_offset</i>	Start edge event.
<i>end_offset</i>	End edge event.

---

The following example defines a period beginning at 4.5 time units before the negative *clk* edge and ending at 4.5 time units after the subsequent positive *clk* edge. Both clock and data events can be edge-triggered and conditional. During this time period, the data signal must not change.

```
(INSTANCE x)
(TIMINGCHECK
 (NOCHANGE (negedge clk) data (4.5) (4.5) ) )
```



## TIMINGENV Keyword and Constructs

**Note:** The TIMINGENV keyword and its constructs are ignored by NC-Verilog. The following syntax is included for completeness. Specifying the TIMINGENV keyword constructs will not generate an error message.

The TIMINGENV keyword associates constraint values with critical paths in the design and provides information about the timing environment in which the circuit operates. Constraint entries provide information about the timing properties that a design is required to have to meet certain design objectives.

The syntax is as follows:

```
(TIMINGENV
  { (PATHCONSTRAINT { "path_name" } start_path {int_path+} end_path
    max_rise max_fall) }

  { (PERIODCONSTRAINT edge_clk max_period { (EXCEPTION (INSTANCE path+))} ) }

  { (SKEWCONSTRAINT port_path max_skew) }

  { (SUM (start_path end_path) { (start_path end_path)+} (start_end_sum)
    {(start_end_sum)+}) }

  { (DIFF (start_path end_path) { (start_path end_path)+} (start_end_diff)
    {(start_end_diff)+}) }

  { (ARRIVAL { (edge port) } port_name early_rise late_rise early_fall
    late_fall) }

  { (DEPARTURE { (edge port) } port_name early_rise late_rise early_fall
    late_fall) }

  { (SLACK port rise_setup fall_setup rise_hold fall_hold {clk_period} )

  { (WAVEFORM port wave_period (edge num1 {num2})+ (edge num1 {num2})+ ) }

)
```

## **PATHCONSTRAINT Keyword**

**Note:** The PATHCONSTRAINT keyword and all other TIMINGENV constructs are ignored by NC-Verilog.

The PATHCONSTRAINT keyword specifies the maximum allowable delay for a path, which is typically identified by the two ports at each end of the path. Path constraints are the critical paths in a design identified during timing analysis. You can also specify intermediate ports to uniquely identify path(s). Layout tools use these constraints to direct the physical design. The syntax is as follows:

```
(PATHCONSTRAINT {"path_name"} start_path {int_path+} end_path max_rise  
max_fall)
```

---

<b>Keyword Argument</b>	<b>Description</b>
<i>path_name</i>	Optional symbolic or actual name of the path.
<i>start_path</i>	Start of port path. Where applicable, a port path can have array index (for example, <i>x.y[3].p</i> ).
<i>int_path</i>	Intermediate points to describe the path. You can have multiple <i>int_path</i> arguments. Where applicable, a port path can have array index (for example, <i>x.y[3].p</i> ).
<i>end_path</i>	End of port path. Where applicable, a port path can have array index (for example, <i>x.y[3].p</i> ).
<i>max_rise</i>	Maximum rise delay between the start and end path points.
<i>max_fall</i>	Maximum fall delay between the start and end path points.

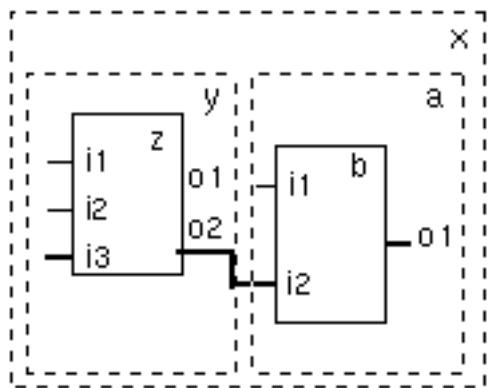
---

The following example shows how to use the PATHCONSTRAINT keyword:

## NC-Verilog Simulator Help

### SDF File Syntax

```
(INSTANCE x)
(TIMINGENV
  (PATHCONSTRAINT y.z.i3 y.z.o2 a.b.o1 (25.1) (15.6)
  )
)
```



The following example shows a cell entry specifying a path constraint.

```
(CELL (CELLTYPE "DFF")
  (INSTANCE a.b.c)
  (TIMINGENV
    (PATHCONSTRAINT y.z.i3 a.b.o1 (25)(15))
  )
)
```

## PERIODCONSTRAINT Keyword

**Note:** The PERIODCONSTRAINT keyword and all other TIMINGENV constructs are ignored by NC-Verilog.

The PERIODCONSTRAINT keyword specifies the maximum time allowable for one complete cycle of the signal. The syntax is as follows:

```
(PERIODCONSTRAINT edge_clk max_period {(EXCEPTION (INSTANCE path+))} )
```

---

Keyword Argument	Description
<i>edge_clk</i>	Edge-triggered clock event.
<i>max_period</i>	Maximum period for complete signal cycle.
<b>EXCEPTION</b>	A list of one or more cell instances to be excluded from the group.
<b>INSTANCE</b>	Cell instance.

---

**Note:** You must specify the PERIODCONSTRAINT keyword at a code-hierarchy level that includes the cell instance that drives the common clock inputs of the flip-flops and any cell instances to be placed in the exception list.

The following example shows how to use the PERIODCONSTRAINT keyword.

```
(INSTANCE x)
(TIMINGENV
  (PERIODCONSTRAINT bufa.y (10)
    (EXCEPTION (INSTANCE dff3)
     )
   )
)
```

## SKEWCONSTRAINT Keyword

**Note:** The SKEWCONSTRAINT keyword and all other TIMINGENV constructs are ignored by NC-Verilog.

The SKEWCONSTRAINT keyword specifies the clock event signal which is constrained against the associated port signals. For example, in a chain of devices such as counters that are connected to the same clock signal, the clock ideally changes at exactly the same time at all counters. However, there is some delay between the change at one device and the change at another. If this delay exceeds the signal skew limit, the data passed along the chain is unreliable. The syntax is as follows:

(SKEWCONSTRAINT *port\_path* *max\_skew*)

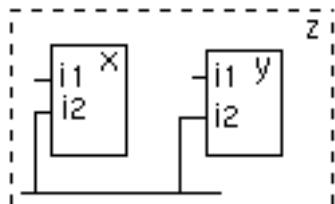
---

<b>Keyword Argument</b>	<b>Description</b>
<i>port_path</i>	Port that drives the net. Where applicable, a port path can have array index (for example, <i>x.y[3].p</i> ).
<i>max_skew</i>	Maximum skew between signals is identified as a design constraint by timing analysis tools. This information can be passed through the SDF file to layout tools to ensure that the physical design is laid out within these constraints.

---

The following example shows how to use the SKEWCONSTRAINT keyword:

```
(INSTANCE z)
(TIMINGENV
  (SKEWCONSTRAINT (posedge i2) (7.5)
  )
)
```



## SUM Keyword

**Note:** The SUM keyword and all other TIMINGENV constructs are ignored by NC-Verilog.

The SUM keyword specifies the maximum sum of two or more path delays. The syntax is as follows:

```
(SUM (start_path end_path) {(start_path end_path)+}  
  (start_end_sum) {(start_end_sum)+}  
)
```

---

Keyword Argument	Description
<i>start_path</i>	Start of port path. Where applicable, a port path can have array index (for example, <i>x.y[ 3 ].p</i> ).
<i>end_path</i>	End of port path. Where applicable, a port path can have array index (for example, <i>x.y[ 3 ].p</i> ).
<i>start_end_sum</i>	Sum of the individual delays associated with each start and end port pair. The sum of all port pair delays must be less than the value of <i>start_end_path</i> .

---

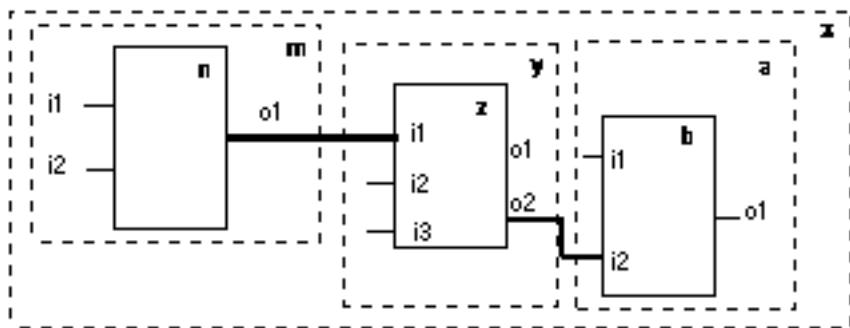
**Note:** You can specify additional paths by using additional pairs of *start\_path* and *end\_path* arguments with corresponding *start\_end\_sum* arguments.

## NC-Verilog Simulator Help

### SDF File Syntax

The following example shows how to use the SUM keyword:

```
(INSTANCE x)
(TIMINGENV
  (SUM (m.n.o1 y.z.i1) (y.z.o2 a.b.i2) (67.3)
    )
)
```



## DIFF Keyword

**Note:** The DIFF keyword and all other TIMINGENV constructs are ignored by NC-Verilog.

The DIFF keyword specifies the maximum allowable difference between the delays of two paths in a design. The syntax is as follows:

```
(DIFF (start_path end_path) {(start_path end_path)+}
      (start_end_diff) {(start_end_diff)+} )
```

Keyword Argument	Description
<i>start_path</i>	Start of port path. Where applicable, a port path can have array index (for example, <i>x.y[3].p</i> ).
<i>end_path</i>	End of port path. Where applicable, a port path can have array index (for example, <i>x.y[3].p</i> ).
<i>start_end_diff</i>	Maximum difference between two path delays. The difference of the individual delays in the two circuit paths must be less than the value of <i>start_end_diff</i> .

**Note:** You can specify additional paths by using additional pairs of *start\_path* and *end\_path* arguments with corresponding *start\_end\_diff* arguments.

## NC-Verilog Simulator Help

### SDF File Syntax

---

The following example shows how to use the DIFF keyword:

```
(INSTANCE x)
(TIMINGENV
  (DIFF (m.n.o1 y.z.i1) (y.z.o2 a.b.i2) (8.3) ) )
```

## ARRIVAL Keyword

**Note:** The ARRIVAL keyword and all other TIMINGENV constructs are ignored by NC-Verilog.

The ARRIVAL keyword specifies the time when a primary input signal is applied during the intended circuit operation. You use this keyword to analyze the timing behavior for a circuit and to compute logic constraints for logic synthesis and layout. The syntax is as follows:

```
(ARRIVAL {(edge port)} port_name early_rise late_rise
           early_fall late_fall)
```

---

Keyword Argument	Description
<i>edge</i>	Either posedge or negedge.
<i>port</i>	The input port from which the time reference is specified. This required primary input signal is a fan-out from a sequential element (usually an active edge of a clock signal).
<i>port_name</i>	The input or inout port for which the arrival time is to be defined. This port must be a primary (external) input of the top-level module.
<i>early_rise</i>	Earliest rise value relative to the time reference. This value must be less than the <i>late_rise</i> value.
<i>late_rise</i>	Latest rise value relative to the time reference. This value must be greater than the <i>early_rise</i> value.
<i>early_fall</i>	Earliest fall value relative to the time reference. This value must be less than the <i>late_fall</i> value.
<i>late_fall</i>	Latest fall value relative to the time reference. This value must be greater than the <i>early_fall</i> value.

---

The following example shows how to use the ARRIVAL keyword. It applies rising transitions at D[15:0] no sooner than 10 and no later than 40 time units after the rising edge of the reference time MCLK. It applies falling transitions no sooner than 12 and no later than 45 time units after the edge.

## NC-Verilog Simulator Help

### SDF File Syntax

---

```
(INSTANCE top)
(TIMINGENV
  (ARRIVAL (posedge MCLK) D[15:0] (10) (40) (12) (45)
  )
)
```

## DEPARTURE Keyword

**Note:** The DEPARTURE keyword and all other TIMINGENV constructs are ignored by NC-Verilog.

The DEPARTURE keyword specifies the time when a primary output signal is applied during the intended circuit operation. You use this keyword to analyze the timing behavior for a circuit and to compute logic constraints for logic synthesis and layout. The syntax is as follows:

```
(DEPARTURE {(edge port)} port_name early_rise late_rise
           early_fall late_fall)
```

---

Keyword Argument	Description
<i>edge</i>	Either posedge or negedge.
<i>port</i>	The input port from which the time reference is specified. This required primary input signal is a fan-out from a sequential element (usually an active edge of a clock signal).
<i>port_name</i>	The output or inout port for which the departure time is to be defined. This port must be a primary (external) output of the top-level module.
<i>early_rise</i>	Earliest rise value relative to the time reference. This value must be less than the <i>late_rise</i> value.
<i>late_rise</i>	Latest rise value relative to the time reference. This value must be greater than the <i>early_rise</i> value.
<i>early_fall</i>	Earliest fall value relative to the time reference. This value must be less than the <i>late_fall</i> value.
<i>late_fall</i>	Latest fall value relative to the time reference. This value must be greater than the <i>early_fall</i> value.

---

The following example shows how to use the DEPARTURE keyword. It applies rising transitions at A[15:0] no sooner than 8 and no later than 20 time units after the rising edge

## NC-Verilog Simulator Help

### SDF File Syntax

---

of the reference time SCLK. It applies falling transitions no sooner than 12 and no later than 34 time units after the edge.

```
(INSTANCE top)
(TIMINGENV
  (DEPARTURE (posedge SCLK) D[15:0] (8) (20) (12) (34)
  )
)
```

## SLACK Keyword

**Note:** The SLACK keyword and all other TIMINGENV constructs are ignored by NC-Verilog.

The SLACK keyword compares the calculated delay over a path to the delay constraints imposed upon the path and determines the available slack or margin in the delay path. Positive slack indicates that the constraints are met with additional time units to spare. Negative slack indicates a failure to construct a circuit according to the constraints. The syntax is as follows:

```
(SLACK port rise_setup fall_setup rise_hold fall_hold {clk_period})
```

---

Keyword Argument	Description
<i>port</i>	Input port that provides the slack or margin information.
<i>rise_setup</i>	Additional rise delay that can be tolerated in all paths ending at the <i>port</i> without causing the design constraint to be violated.
<i>fall_setup</i>	Additional fall delay that can be tolerated in all paths ending at the <i>port</i> without causing the design constraint to be violated.
<i>rise_hold</i>	Reduction of rise delay that can be tolerated in all paths ending at the <i>port</i> without causing the design constraint to be violated.
<i>fall_hold</i>	Reduction of fall delay that can be tolerated in all paths ending at the <i>port</i> without causing the design constraint to be violated.
<i>clk_period</i>	Optionally represents the clock period on which the slack or margin values are based. The clock period is specified by the WAVEFORM keyword construct.

---

**Note:** You can specify multiple SLACK keyword constructs for the same port. They are distinct as long as the value of *clk\_period* is different.

## NC-Verilog Simulator Help

### SDF File Syntax

---

The following example shows that the signal arrives at port `macro.AOI6.B` in time to meet the setup time requirement of a flip-flop down the path with 3 time units to spare. Therefore, the delay of any and all paths leading to port `macro.AOI6.B` can be increased by an additional 3 time units without violating a setup requirement. The example also shows that the delay of any data paths leading to port `macro.AOI6.B` can be decreased by 7 time units without violating a hold requirement.

```
(CELL
  (CELLTYPE "cpu")
  (INSTANCE macro.AOI6)
  (TIMINGENV
    (SLACK B (3) (3) (7) (7)
    )
  )
)
```

## WAVEFORM Keyword

**Note:** The WAVEFORM keyword and all other TIMINGENV constructs are ignored by NC-Verilog.

The WAVEFORM keyword specifies a periodic waveform that is applied to a circuit during its intended operation. Typically, you use this to define a clock signal. You use this keyword to analyze the timing behavior for a circuit and to compute logic constraints for logic synthesis and layout. The syntaxes are as follows:

```
(WAVEFORM port wave_period (posedge num1 {num2}) (negedge num1 {num2}) )
(WAVEFORM port wave_period (negedge num1 {num2}) (posedge num1 {num2}) )
```

**Note:** Specifying `posedge` or `negedge` first determines the direction of the transition.

---

Keyword Argument	Description
<i>port</i>	Input or inout port where the waveform appears.
<i>wave_period</i>	Number that specifies the waveform that repeats indefinitely at the interval.
<i>num1</i>	When specified alone, defines the transition offset. When specified with <i>num2</i> , defines the beginning of the uncertainty region in which the transition takes place.
<i>num2</i>	Defines the end of the uncertainty region in which the transition takes place.

---

## NC-Verilog Simulator Help

### SDF File Syntax

---

If the port is not a primary input of the circuit (that is, if it is driven by the output of some other circuit element in the scope of the analysis), then the signal driven in the circuit should be ignored and the specified waveform should replace it in the analysis.

The following example shows the specification of a waveform of period 15 to be applied to port `top.clka`. Within each period, a rising edge occurs somewhere between 0 and 2 and a falling edge somewhere between 5 and 7.

```
(CELL (CELLTYPE "cpu")
      (INSTANCE top)
      (TIMINGENV (WAVEFORM clka 15 (posedge 0 2) (negedge 5 7))
      )
)
```

The following example shows the specification of a waveform of period 25 to be applied to port `top.clkb`. Within each period, a falling edge occurs at 0, a rising edge at 5, a falling edge at 10, and a rising edge at 15.

```
(CELL (CELLTYPE "cpu")
      (INSTANCE top)
      (TIMINGENV (WAVEFORM clkb 25 (negedge 0) (posedge 5)
      (negedge 10) (posedge 15))
      )
)
```

The following example shows negative numbers in defining a waveform.

```
(CELL (CELLTYPE "cpu")
      (INSTANCE top)
      (TIMINGENV (WAVEFORM clkb 50 (negedge -10) (posedge 20))
      )
)
```

## OVI SDF Specification Version Differences

The SDF annotator supports multiple versions of the OVI SDF specifications. The following sections summarize the major differences between the versions.

### SDF Version 1.\* Constructs

Specify 1.\* for any version prior to 2.0. If no SDFVERSION is specified, version 1.0 is used by default. Version 1.\* specifies the conditional TIMINGCHECK construct differently than later versions, as follows:

```
(COND ... (timing_check ...))
```

This implies that you can supply a single condition to the timing check. By contrast, in SDF version 2.0 or greater, you can match one or more timing check terminals, if more than one is present.

### SDF Version 2.\* Constructs

The following constructs are specific to the 2.\* versions of the SDF standard:

- The GLOBALPATHPULSE construct is supported. This was changed to PATHPULSEPERCENT in Version 3.0.
- You can optionally specify the TIMESCALE keyword in Version 2.0; in Version 2.1, the TIMESCALE keyword is required. The default for TIMESCALE is 1 nanosecond (ns).
- The timing constraints (PATHCONSTRAINT, SUM, DIFF, SKEWCONSTRAINT) are allowed within the TIMINGCHECK construct.

### SDF Version 3.\* Constructs

The following constructs are specific to the 3.\* versions of the SDF standard:

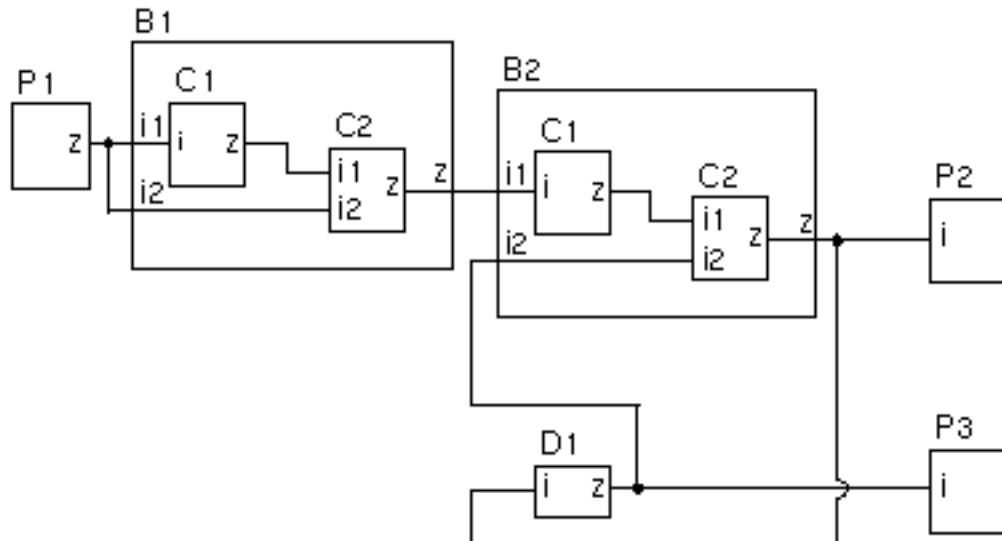
- The PATHPULSEPERCENT keyword replaces the 2.\* GLOBALPATHPULSE keyword but the functionality is the same.
- Consecutive INSTANCE constructs are not allowed. In addition, the INSTANCE construct is allowed only in the CELL header.
- You can specify 12 delay values, the extra 6 delay values being x transition delays.
- The CONDELSE construct is supported.
- The RETAIN construct is supported in IOPATH, COND IOPATH, and CONDELSE IOPATH constructs.

- The REMOVAL timing check is allowed.
- The RECOVERY construct allows a single limit only, and does not allow use of the SCOND or CCOND constructs. Two-limit recoveries are annotated using the RECREM construct.
- The NETDELAY construct is no longer supported.
- The RECREM construct is supported.
- You can specify timing constraint constructs only in the TIMINGENV construct. In addition the new TIMINGENV entries (ARRIVAL, DEPARTURE, SLACK, WAVEFORM) are supported.

## SDF File Examples

### Example 1

The SDF file example that follows is based on the following schematic.



```
(DELAYFILE
  (SDFVERSION "2.1")
  (DESIGN "system")
  (DATE "Saturday December 14 08:30:33 PST 1996")
  (VENDOR "Cadence")
  (PROGRAM "delay_calc")
  (VERSION "1.5"))
```

## NC-Verilog Simulator Help

### SDF File Syntax

---

```
(DIVIDER /)
(VOLTAGE 4.5:5.0:5.5)
(PROCESS "worst")
(TEMPERATURE 55:85:125)
(TIMESCALE 1ns)

(CELL (CELLTYPE "system") (INSTANCE block_1)
  (DELAY (ABSOLUTE
    (INTERCONNECT P1/z B1/C1/i (.145:::145) (.125:::125))
    (INTERCONNECT P1/z B1/C2/i2 (.135:::135) (.130:::130))
    (INTERCONNECT B1/C1/z B1/C2/i1 (.095:::095) (.095:::095))
    (INTERCONNECT B1/C2/z B2/C1/i (.145:::145) (.125:::125))
    (INTERCONNECT B2/C1/z B2/C2/i1 (.075:::075) (.075:::075))
    (INTERCONNECT B2/C2/z P2/i (.055:::055) (.075:::075))
    (INTERCONNECT B2/C2/z D1/i (.255:::255) (.275:::275))
    (INTERCONNECT D1/z B2/C2/i2 (.155:::155) (.175:::175))
    (INTERCONNECT D1/z P3/i (.155:::155) (.130:::130)))))

(CELL (CELLTYPE "INV") (INSTANCE B1/C1)
  (DELAY (ABSOLUTE
    (IOPATH i z (.345:::345) (.325:::325)))))

(CELL (CELLTYPE "OR2") (INSTANCE B1/C2)
  (DELAY (ABSOLUTE
    (IOPATH i1 z (.300:::300) (.325:::325))
    (IOPATH i2 z (.300:::300) (.325:::325)))))

(CELL (CELLTYPE "INV") (INSTANCE B2/C1)
  (DELAY (ABSOLUTE
    (IOPATH i z (.345:::345) (.325:::325)))))

(CELL (CELLTYPE "AND2") (INSTANCE B2/C2)
  (DELAY (ABSOLUTE
    (IOPATH i1 z (.300:::300) (.325:::325))
    (IOPATH i2 z (.300:::300) (.325:::325)))))

(CELL (CELLTYPE "INV") (INSTANCE D1)
  (DELAY (ABSOLUTE
    (IOPATH i z (.380:::380) (.380:::380)))))

) // end delayfile
```

## Example 2

This example shows how you can use the COND construct with the IOPATH and TIMINGCHECK constructs.

```
(DELAYFILE
  (SDFVERSION "2.0")
  (DESIGN "top")
  (DATE "Nov 12, 1996 11:30:10")
  (VENDOR "Cool New Tools")
  (PROGRAM "Delay Obfuscator")
  (VERSION "v1.0")
  (DIVIDER .)
  (VOLTAGE :5:)
  (PROCESS "typical")
  (TEMPERATURE :25:)
  (TIMESCALE 1ns)
  (CELL (CELLTYPE "CDS_GEN_FD_P_SD_RB_SB_NO")
    (INSTANCE top.ff1)
    (DELAY
      (ABSOLUTE (COND (TE == 0 && RB == 1 && SB == 1)
        (IOPATH (posedge CP) Q (2:2:2) (3:3:3))))
      (ABSOLUTE (COND (TE == 0 && RB == 1 && SB == 1)
        (IOPATH (posedge CP) QN (4:4:4) (5:5:5))))
      (ABSOLUTE (COND (TE == 1 && RB == 1 && SB == 1)
        (IOPATH (posedge CP) Q (6:6:6) (7:7:7))))
      (ABSOLUTE (COND (TE == 1 && RB == 1 && SB == 1)
        (IOPATH (posedge CP) QN (8:8:8) (9:9:9))))
      (ABSOLUTE
        (IOPATH (negedge RB) Q (1:1:1) (1:1:1)))
      (ABSOLUTE
        (IOPATH (negedge RB) QN (1:1:1) (1:1:1)))
      (ABSOLUTE
        (IOPATH (negedge SB) Q (1:1:1) (1:1:1)))
      (ABSOLUTE
        (IOPATH (negedge SB) QN (1:1:1) (1:1:1)))
    ) // end delay
    (DELAY
      (ABSOLUTE (PORT D (0:0:0) (0:0:0) (5:5:5)))
      (ABSOLUTE (PORT CP (0:0:0) (0:0:0) (0:0:0)))
      (ABSOLUTE (PORT RB (0:0:0) (0:0:0) (0:0:0)))
      (ABSOLUTE (PORT SB (0:0:0) (0:0:0) (0:0:0)))
    )
  )
)
```

## NC-Verilog Simulator Help

### SDF File Syntax

---

```
(ABSOLUTE (PORT TI (0:0:0) (0:0:0) (0:0:0)))
(ABSOLUTE (PORT TE (0:0:0) (0:0:0) (0:0:0)))
) // end delay
(TIMINGCHECK
  (COND D_ENABLE (SETUP D (posedge CP) (1:1:1)))
  (COND D_ENABLE (HOLD D (posedge CP) (1:1:1)))
  (COND TI_ENABLE (SETUPHOLD TI (posedge CP)) (1:1:1) (1:1:1))
  (COND ENABLE (WIDTH (posedge CP) (1:1:1)))
  (COND ENABLE (WIDTH (negedge CP) (1:1:1)))
  (WIDTH (negedge SB) (1:1:1))
  (WIDTH (negedge RB) (1:1:1))
  (COND SB (RECOVERY (posedge RB) (negedge CP) (1:1:1)))
  (COND RB (RECOVERY (posedge SB) (negedge CP) (1:1:1)))
) // end timingcheck
) // end cell
) // end delayfile
```

### Example 3

This example shows how State Dependent Path Delays (SDPDs) can be annotated using COND and IOPATH constructs.

```
(DELAYFILE
  (SDFVERSION "2.0")
  (DESIGN "top")
  (DATE "May 12, 1996 17:25:18")
  (VENDOR "Slick Trick Systems")
  (PROGRAM "Viability Tester")
  (VERSION "v3.0")
  (DIVIDER .)
  (VOLTAGE :5:) (PROCESS "typical") (TEMPERATURE :25:)
  (TIMESCALE 1ns)
  (CELL (CELLTYPE "XOR2") (INSTANCE top.x1)
    (DELAY
      (INCREMENT (COND i1 (IOPATH i2 o1 (2:2:2) (2:2:2))))
      (INCREMENT (COND i2 (IOPATH i1 o1 (2:2:2) (2:2:2))))
      (INCREMENT (COND ~i1 (IOPATH i2 o1 (3:3:3) (3:3:3))))
      (INCREMENT (COND ~i2 (IOPATH i1 o1 (3:3:3) (3:3:3))))
    )
  )
)
```

## C

---

# System Task Support in the NC-Verilog Simulator

---

The NC-Verilog simulator supports many Verilog-XL system tasks. However, some system tasks are supported as Tcl (interactive) commands, rather than as system tasks in the language, and some system tasks are not supported at all. This appendix tells you which system tasks are supported in the NC-Verilog simulator and which are not supported.

**Note:** In the table, a dash ( — ) means “not applicable.”

System Task	Language Support	Tcl Command
.	—	<a href="#">run</a>
,	—	<a href="#">run -step</a> (But does not trace)
;	—	<a href="#">run -step</a>
:	—	No
<i>number</i>	—	<a href="#">!number</a>
<a href="#">-number</a>	—	No
\$async\$array	Yes	—
\$async\$plane	Yes	—
\$async\$nand\$array	Yes	—
\$async\$nand\$plane	Yes	—
\$async\$nor\$array	Yes	—
\$async\$nor\$plane	Yes	—
\$async\$or\$array	Yes	—
\$async\$or\$plane	Yes	—
\$bitstoreal	Yes	No

## NC-Verilog Simulator Help

### System Task Support in the NC-Verilog Simulator

---

<b>System Task</b>	<b>Language Support</b>	<b>Tcl Command</b>
\$cleartrace	No	No
\$clockdef	Yes	No.
		Can be done with a Tcl script.
\$compare	Yes	No
\$countdrivers	Yes	No.
		Can be done with a Tcl script. See “ <a href="#">Example Tcl Script for \$countdrivers</a> ” on page 1065 for an example script.
\$db_breakaftertime	No	<u>run</u> -relative <i>time</i>
\$db_breakatline	No	<u>stop</u> -line <i>line_number</i>
\$db_breakbeforetime	No	<u>run</u> -absolute <i>time</i>
\$db_breakonceatline	No	<u>stop</u> -line <i>line_number</i> -delbreak 1
\$db_breakonceonnegedge	No	<u>stop</u> -condition {# <i>signame</i> == 1'b0} -delbreak 1
\$db_breakonceonposedge	No	<u>stop</u> -condition {# <i>signame</i> == 1'b1} -delbreak 1
\$db_breakoncewhen	No	<u>stop</u> -object <i>signame</i> -delbreak 1
\$db_breakonnegedge	No	<u>stop</u> -condition {# <i>signame</i> == 1'b0}
\$db_breakonposedge	No	<u>stop</u> -condition {# <i>signame</i> == 1'b1}
\$db_breakwhen	No	<u>stop</u> -object <i>signame</i>
\$db_cleartrace	No	No
\$db_deletebreak	No	<u>stop</u> -delete <i>stop_name</i>
\$db_deletefocus	No	No

## NC-Verilog Simulator Help

### System Task Support in the NC-Verilog Simulator

---

<b>System Task</b>	<b>Language Support</b>	<b>Tcl Command</b>
\$db_disablebreak	No	<u>stop</u> -disable <i>stop_name</i>
\$db_disablefocus	No	No
\$db_enablebreak	No	<u>stop</u> -enable <i>stop_name</i>
\$db_enablefocus	No	No
\$db_help	No	<u>help</u>
\$db_setfocus	No	No
\$db_settrace	No	No
\$db_showbreak	No	<u>stop</u> -show
\$db_showfocus	No	No
\$db_steptime	No	<u>run</u> -step
\$deposit	Yes	<u>deposit</u> <i>signame</i> = <i>value</i>
\$disable_warnings	Yes	No
\$display	Yes	puts [concat " <i>string</i> " [eval value <i>formats objects</i> ]] See " <a href="#">Example Tcl Script for \$display</a> " on page 1066 for an example Tcl script for formatted display.
\$displayh	Yes	puts [concat " <i>string</i> " [eval value %h <i>objects</i> ]]
\$displayb	Yes	puts [concat " <i>string</i> " [eval value %b <i>objects</i> ]]
\$displayo	Yes	puts [concat " <i>string</i> " [eval value %o <i>objects</i> ]]
\$dist_chi_square	Yes	No
\$dist_erlang	Yes	No
\$dist_exponential	Yes	No
\$dist_normal	Yes	No
\$dist_poisson	Yes	No
\$dist_t	Yes	No

## NC-Verilog Simulator Help

### System Task Support in the NC-Verilog Simulator

---

<b>System Task</b>	<b>Language Support</b>	<b>Tcl Command</b>
\$dist_uniform	Yes	No
\$dumpall	Yes	<u>probe</u> -all -database <i>dbase_name</i>
\$dumpfile	Yes	<u>database</u> -vcd <i>dbase_name</i>
\$dumpflush	Yes	No
\$dumplimit	Yes	No.
		Can be done with a Tcl script.
\$dumpoff	Yes	<u>database</u> -disable <i>dbase_name</i>
\$dumpon	Yes	<u>database</u> -enable <i>dbase_name</i>
\$dumpports	Yes	No
\$dumpports_close	Yes	No
\$dumpvars	Yes	<u>probe</u> -database <i>dbase_name</i> -depth <i>level</i>
\$enable_warnings	Yes	No
\$eventcond	No	—
\$fclose	Yes	close <i>file_id</i>
		Used with open.
\$fdisplay	Yes	puts [ <i>file_id</i> ] [concat " <i>string</i> " [eval value formats objects]]]
		Used with open.
\$finish	Yes	<u>finish</u>
\$fmonitor	Yes	script... (Using opened file)
\$fopen	Yes	open <i>filename access_mode</i>

## NC-Verilog Simulator Help

### System Task Support in the NC-Verilog Simulator

---

<b>System Task</b>	<b>Language Support</b>	<b>Tcl Command</b>
\$fstroke	Yes	<code>puts [file_id] [concat "string" [eval value formats objects]]</code>
		Used with <code>open</code> .
\$fwrite	Yes	<code>puts [file_id] [concat "string" [eval value formats objects]]</code>
		Used with <code>open</code> .
\$getpattern	Yes	—
\$gr_regs	No	If you invoke <i>ncsim</i> with <code>-gui</code> , equivalent functionality is provided by SimVision Waveform Viewer.
\$gr_waves	No	If you invoke <i>ncsim</i> with <code>-gui</code> , equivalent functionality is provided by SimVision Waveform Viewer.
\$history	No	<u><a href="#">history</a></u>
\$hold	Yes	—
\$incpattern_read	Yes	No
\$incpattern_write	No	No
\$incsave	No	No
\$input	No	<u><a href="#">source tcl_filename</a></u>
\$itor	Yes	<code>expr double([eval value %g signature])</code>
\$keepcommands	No	Default (no way to turn off)
\$key	No	No. Use <i>ncsim</i> <code>-keyfile</code> option.
\$list	No	<u><a href="#">scope -list</a></u>
\$list_forces	No	No  <u><a href="#">scope -describe</a></u> shows this information along with a lot of other information.

## NC-Verilog Simulator Help

### System Task Support in the NC-Verilog Simulator

---

<b>System Task</b>	<b>Language Support</b>	<b>Tcl Command</b>
\$listcounts	No	No  This information can be obtained with the simulator code coverage feature.
\$log	Yes	No. Use <code>ncsim -logfile</code> option.
\$monitor	Yes	<code>probe -screen</code>  See " <a href="#">Example Tcl Script for \$display</a> " on page 1066 for an example Tcl script.
\$monitoroff	Yes	Can be done with a Tcl script. See " <a href="#">Example Tcl Script for \$display</a> " on page 1066 for an example.
\$monitoron	Yes	Can be done with a Tcl script. See " <a href="#">Example Tcl Script for \$display</a> " on page 1066 for an example.
\$noshowcancelled	No	—
\$nochange	Yes	—
\$noeventcond	No	—
\$nokeepcommands	No	No. See <a href="#">\$keepcommands</a> .
\$nokey	No	No. Use <code>ncsim -nokey</code> option.
\$nolog	Yes	No. Use <code>ncsim -nolog</code> option.
\$period	Yes	—
\$printtimescale	Yes	<code>set time_scale</code>
\$pulse_e_style_ondetect	No	—
\$pulse_e_style_onevent	No	—
\$q_add	Yes	No
\$q_exam	Yes	No
\$q_full	Yes	No
\$q_initialize	Yes	No
\$q_remove	Yes	No

## NC-Verilog Simulator Help

### System Task Support in the NC-Verilog Simulator

---

<b>System Task</b>	<b>Language Support</b>	<b>Tcl Command</b>
\$random	Yes	No
\$readmemb	Yes. See “ <a href="#">The \$readmemb and \$readmemh System Tasks</a> ” on page 1069 for more information.	No. Can be done with a Tcl script. See “ <a href="#">Example Tcl Script for \$readmem</a> ” on page 1070 for an example.
\$readmemh	Yes. See “ <a href="#">The \$readmemb and \$readmemh System Tasks</a> ” on page 1069 for more information.	No. Can use the same script shown in “ <a href="#">Example Tcl Script for \$readmem</a> ” on page 1070.
\$realtime	Yes	<a href="#"><u>time auto</u></a>
\$realtobits	Yes	No
\$recovery	Yes	—
\$reportfile	No	No
\$reportprofile	No	No
\$reset	No	<a href="#"><u>reset</u></a>
\$reset_count	No	No. Can be done with a Tcl script. See “ <a href="#">Example Tcl Script for \$reset_count</a> ” on page 1072 for an example.
\$reset_value	No	No. Can be done with a Tcl script. See “ <a href="#">Example Tcl Script for \$reset_value</a> ” on page 1073 for an example.

## NC-Verilog Simulator Help

### System Task Support in the NC-Verilog Simulator

---

<b>System Task</b>	<b>Language Support</b>	<b>Tcl Command</b>
\$restart	No	<u>restart</u> <i>snapshot_name</i>
\$rtoi	Yes	<u>expr</u> <u>int</u> ([ <i>value</i> <i>real_sig_name</i> ])
\$save	No	<u>save</u> <i>snapshot_name</i>
\$scale	Yes	No
\$scope	Yes	<u>scope</u> -set <i>scope</i>
\$sdf_annotate	Yes	No
\$setattrace	No	No
\$setup	Yes	—
\$setuphold	Yes	—
\$shm_open	Yes	<u>database</u> -shm <i>dbase_name</i>
\$shm_probe	Yes	<u>probe</u> -database <i>dbase_name</i> <i>objects</i>
\$showcancelled	No	—
\$showallinstances	No	No. Can be done with a PLI routine.
\$showexpandednets	No	No. Information is provided in the elaborator log file. Write a PLI routine to get a list of expanded nets.
\$showmodes	No	No. Not all modes are supported yet.
\$shownonxl	No	—
\$showportsnotcollapsed	No	No

## NC-Verilog Simulator Help

### System Task Support in the NC-Verilog Simulator

---

<b>System Task</b>	<b>Language Support</b>	<b>Tcl Command</b>
\$showscopes	No	<u>scope</u> -show
		The <u>scope</u> -show command does not display the entire hierarchy. See “ <a href="#">Example Tcl Script for \$showscopes</a> ” on page 1074 for a recursive script that will recurse the hierarchy if an argument is passed. This is similar to \$showscopes(1);.
\$showvariables	No	<u>describe</u> variable
\$showvars	No	<u>describe</u> variable
\$skew	Yes	—
\$sreadmemb	Yes	No
\$sreadmemh	Yes	No
\$startprofile	No	No
\$stime	Yes	<u>time</u>
\$stop	Yes	<u>stop</u> -time <i>time</i> or <u>run</u> <i>time</i>
\$stopprofile	No	No
\$strobe	Yes	No
\$strobe_compare	Yes	No
\$sync\$array	Yes	—
\$sync\$plane	Yes	—
\$sync\$nand\$array	Yes	—
\$sync\$nand\$plane	Yes	—
\$sync\$nor\$array	Yes	—
\$sync\$nor\$plane	Yes	—
\$sync\$or\$array	Yes	—
\$sync\$or\$plane	Yes	—

## NC-Verilog Simulator Help

### System Task Support in the NC-Verilog Simulator

---

System Task	Language Support	Tcl Command
\$test\$plusargs	Yes	No. Can be done with a PLI routine. See " <a href="#">Example PLI Routine for \$test\$plusargs</a> " on page 1073 for an example.
\$time	Yes	<a href="#"><u>time</u></a>
\$timeformat	Yes	No
\$width	Yes	—
\$write	Yes	<a href="#"><u>puts</u></a>
\$writememb	Yes	
\$writememh	Yes	

## Example Tcl Script for \$countdrivers

The following example Tcl script does not deal with forced drivers. Arguments are optional.

```
proc countdrivers {net_name args} {
    upvar [lindex $args 0] forced
    upvar [lindex $args 1] total
    upvar [lindex $args 2] num0
    upvar [lindex $args 3] num1
    upvar [lindex $args 4] numx

    set drvinfo [split [drivers $net_name] "\n"]
    set total 0
    set tmpx 0
    set tmp1 0
    set tmp0 0
    for {set i 1} {$i < [expr [llength $drvinfo] - 1]} {incr i} {
        set tmpline [lindex $drvinfo $i]
        set linelist [split $tmpline " "]
        set drvval [lindex $linelist 3]
        if {$i == 1} {
            if {[lindex $linelist 5] == "force"} {
                set forced 1
                continue
            } else {
                set forced 0
            }
        }
        switch $drvval {
            StX {incr tmpx}
            St0 {incr tmp0}
            St1 {incr tmp1}
        }
    }

    set total [expr $tmpx + $tmp1 + $tmp0]
    set num0 $tmp0
    set num1 $tmp1
    set numx $tmpx
    if {$total > 1} {return 1}
    else {return 0}
}
```

## Example Tcl Script for \$display

The scripts `display` and `monitor` shown in this section use the `format_string` script to handle a formatting string of the form:

```
"in1 = %b bus = %h"
```

This works like `$monitor`. `time` is a special object that returns simulation time. For example:

```
monitor "%t: in1 = %b busin = %h busout = %d" time in1 busin busout
```

Also included are scripts for `monitoron` and `monitoroff`.

```
proc format_string {strname vars} {
    set tmpstr $strname
    set argcnt 0
    while {[set index [string first % $tmpstr]] != -1} {
        set fc [string range $tmpstr $index [expr ($index + 1)]]
        if {[lindex $vars $argcnt] == ""} {
            puts "Formatting error... no verilog object given for
                  format character \"$fc\""
            return -1
        }
        switch $fc {
            %b { regsub $fc $tmpstr [value %b [lindex $vars
                $argcnt]] tmpstr incr argcnt
            }
            %c { regsub $fc $tmpstr [value %c [lindex $vars
                $argcnt]] tmpstr incr argcnt
            }
            %s { regsub $fc $tmpstr [value %s [lindex $vars
                $argcnt]] tmpstr incr argcnt
            }
            %o { regsub $fc $tmpstr [value %o [lindex $vars
                $argcnt]] tmpstr incr argcnt
            }
            %h { regsub $fc $tmpstr [value %h [lindex $vars
                $argcnt]] tmpstr incr argcnt
            }
            %x { regsub $fc $tmpstr [value %x [lindex $vars
                $argcnt]] tmpstr incr argcnt
            }
        }
    }
}
```

## NC-Verilog Simulator Help

### System Task Support in the NC-Verilog Simulator

---

```
%d { regsub $fc $tmpstr [value %d [lindex $vars
    $argcount]] tmpstr incr argcoun
}
%t { if {[lindex $vars $argcount] == "time"} {
    regsub $fc $tmpstr [time module] tmpstr
} elseif {[lindex $vars $argcount] == "realtime"} {
    regsub $fc $tmpstr [time auto] tmpstr
} else {
    regsub $fc $tmpstr [value %t [lindex $vars
    $argcount]] tmpstr
}
incr argcoun
}
%v { regsub $fc $tmpstr [value %v [lindex $vars
    $argcount]] tmpstr incr argcoun
}
%e { regsub $fc $tmpstr [value %e [lindex $vars
    $argcount]] tmpstr incr argcoun
}
%f { regsub $fc $tmpstr [value %f [lindex $vars
    $argcount]] tmpstr incr argcoun
}
%g { regsub $fc $tmpstr [value %g [lindex $vars
    $argcount]] tmpstr incr argcoun
}
}
}
return $tmpstr
}

proc display {formatstring args} {
set tmpstr [format_string $formatstring $args]
if {$tmpstr == -1} {
    puts "Error in display command"
    return
}
puts "$tmpstr"
}
```

## NC-Verilog Simulator Help

### System Task Support in the NC-Verilog Simulator

---

```
proc mon_display {formatstring args} {
    set tmpstr [format_string $formatstring $args]
    if {$tmpstr == -1} {
        puts "Error in monitor command"
        return
    }
    puts "$tmpstr"
}

proc monitor {formatstring args} {
    global default_mon
    foreach i $args {
        if {$i != "time" && $i != "realtime"} {
            set tmpvar [stop -object $i -execute "mon_display
                \"$formatstring\" \"$args\" -silent -continue"]
            set tmpvar [string trim [lindex [split $tmpvar " "] 2]
                "\n"]
            lappend monlist $tmpvar
        }
    }
    set default_mon $monlist
    return $monlist
}

proc monitoroff {{monlist -1}} {
    global default_mon
    global last_mon
    if {$monlist == -1} {
        puts "No monitor list give...disabling the last monitor"
        set monlist $default_mon
    }
    foreach i $monlist {
        stop -disable $i
    }
    set last_mon $monlist
}
```

```
proc monitoron {{monlist -1}} {
    global default_mon
    if {$monlist == -1} {
        puts "No monitor list give...enabling the last disabled
              monitor"
        set monlist $last_mon
    }
    foreach i $monlist {
        stop -enable $i
    }
}
```

## The **\$readmemb** and **\$readmemh** System Tasks

The IEEE standard says that, if no addressing information is specified in the system task, and if no address specification appears in the data file, the default start address for loading the memory is the left-hand address given in the declaration of the memory.

For NC-Verilog, the default start address is the lowest address given in the declaration. Data is read into the memory from the lowest address to the upper address. This matches the behavior of Verilog-XL. For example, given the following memory declaration:

```
reg[7:0] mem{256:1};
```

NC-Verilog will load the memory starting with the lowest address (in this case, the address on the right).

To match the behavior specified in the standard, you can declare the memory with the lowest address on the left, or you can specify both the starting and ending address to **\$readmemb** or **\$readmemh**.

## Example Tcl Script for \$readmem

The following example Tcl script for \$readmem functionality handles all formats.

```
proc readmemb {filename memname {startadd -1} {endadd -1}} {
    return [readmem $filename $memname $startadd $endadd 'b']
}

proc readmemh {filename memname {startadd -1} {endadd -1}} {
    return [readmem $filename $memname $startadd $endadd 'h']
}

proc readmemo {filename memname {startadd -1} {endadd -1}} {
    return [readmem $filename $memname $startadd $endadd 'o']
}

proc readmemd {filename memname {startadd -1} {endadd -1}} {
    return [readmem $filename $memname $startadd $endadd 'd']
}

proc readmem {filename memname startaddr endaddr format} {
    set fname [open $filename r]
    set depth [lindex [split [describe $memname] " "] 2]
    set laddr [lindex [split $depth {[ } :}] 1]
    set raddr [lindex [split $depth {[ } :}] 2]
    set lowest [expr ($laddr<$raddr)?$laddr:$raddr]
    set highest [expr ($laddr<$raddr)?$raddr:$laddr]

    if {$startaddr != -1} {
        set tmpstart [expr vdec($startaddr)]
        set startchar [string first d $tmpstart]
        set tmpstart [string range $tmpstart [expr ($startchar + 1)] end]

        if {$endaddr != -1} {
            set tmpend [expr vdec($endaddr)]
            set startchar [string first d $tmpend]
            set tmpend [string range $tmpend [expr ($startchar + 1)] end]
            set start [expr ($tmpend<$tmpstart)?$tmpend: $tmpstart]
            set tmpend [expr ($tmpend<$tmpstart)?$tmpstart:$tmpend]
            set tmpstart $start
        } else {
            set tmpend $highest
        }
    } else {
        set tmpstart $lowest
    }
}
```

## NC-Verilog Simulator Help

### System Task Support in the NC-Verilog Simulator

---

```
        set tmpend $highest
    }
if {$tmpstart < $lowest} {
    puts "Error: start address is out of range (min address is $lowest)"
    return -1
}
set lowest $tmpstart
if {$tmpend > $highest} {
    puts "Error: end address is out of range (max address is $highest)"
    return -1
}
set highest $tmpend
set addr $lowest
puts "*** Reading memfile $filename, address locations $lowest to $highest"
set inblock 0
while {[gets $fname vecin] >= 0} {
    if {$inblock == 1} {
        if {[regex {/*} $vector]} {
            set inblock 0
        }
        continue
    } else {
        if {[regex {/*} $vector]} {
            if {[regex {/*} $vector]} {continue}
            set inblock 1
            continue
        }
    }
    if {[regex {\w*//}] $vector} {
        continue
    }
    set vector [string trim $vector]
    if {[string length $vector] == 0} { continue }
    set vector [string trim $vecin]
    if {[regexp {@[a-zA-Z0-9]+$} $vector]} {
        set addr [expr vdec('h[string trim $vector @]')]
        set startchar [string first d $addr]
        set addr [string range $addr [$startchar + 1]] end]
        continue
    }
    if {$addr > $highest} {
```

```
        break
    }
    if {$addr < $lowest} {
        incr addr
        continue
    }
puts "deposit $memname[$addr] $format$vector"
deposit $memname[$addr] $format$vector
incr addr
}
close $fname
return 0
}
```

## Example Tcl Script for \$reset\_count

You can use the following Tcl example script instead of the \$reset\_count system task:

```
proc reset_sim {val} {
    global reset_cnt
    global reset_val
    set reset_val $val
    if {[info exists reset_cnt]} {
        incr reset_cnt
    } else {
        set reset_cnt 1
    }
    reset
}
```

## Example Tcl Script for \$reset\_value

You can use the following example Tcl script instead of the \$reset\_value system task:

```
proc reset_value {} {
    global reset_val
    if {[info exists reset_val]} {
        return $reset_val
    } else {
        return 0
    }
}
```

## Example PLI Routine for \$test\$plusargs

```
int testplusargs () {
    char testval [256];
    char *retval;
    if(tf_nump()!=1) {
        tf_putstr(0, 0);
        return(-1);
    }
    strcpy(testval, tf_getcstringp(1));
    retval = mc_scan_plusargs(testval);
    if(!retval) tf_putstr(0, 0);
    else tf_putstr(0, 1);
}
/** for the veriuser.c file ***/
{ userfunction, 0, 0, 0, testplusargs, 0, "$testplusargs" },
```

To use this routine from Tcl mode, use the call command:

```
ncsim> call testplusargs {"string"}
```

## Example Tcl Script for \$showscopes

The Tcl `scope -show` command does not display the entire hierarchy. The script shown in this section is a recursive script. This script will recurse the hierarchy if an argument is passed. This is similar to `$showscopes(1);`.

```
proc showscopes {{all 0}} {
    if {$all == 0} {
        puts [scope -show]
        return
    }
    puts "Directory of scopes at current scope level:"
    set tmp [scope -show]
    set curscope [string range $tmp [expr ([string first "Current
        scope" $tmp] + 18)] [expr ([string last ".")" $tmp] - 1)]]
    printsopes $curscope "      "
    puts "\nCurrent scope is ($curscope)"
    puts "Highest level modules:"
    puts -nonewline [string range $tmp [expr ([string first
        "modules:\n" $tmp] + 9)] end]
}
proc printsopes {currentscope tabs} {
    scope -set $currentscope
    set scopes [split [scope -describe] "\n"]
    foreach i $scopes {
        set name [string range $i 0 [expr ([string first .. $i] - 1)]]
        if {[string match "*instance *" $i]} {
            if {[string match "* module *" $i]} {
                set j [string range $i [expr ([string first module $i] + 7)] end]
                puts -nonewline "$tabs"
                puts "module ($j), instance ($name)"
                printsopes "$currentscope.$name" "      $tabs"
                scope -set $currentscope
            } elseif {[string match "* gate *" $i]} {
                set j [string range $i [expr ([string first "" $i] + 1)] [expr
                    ([string first " gate " $i] - 2)]]
            } elseif {[string match "* UDP *" $i]} {
                set j [string range $i [expr ([string first "UDP" $i] + 4)] [expr
                    ([string first "=" $i] -1)]]
            }
        } elseif {[string match "*task*" $i]} {
            puts -nonewline "$tabs"
```

## NC-Verilog Simulator Help

### System Task Support in the NC-Verilog Simulator

---

```
    puts "task ($name)"
    printsopes "$currentscope.$name" "      $tabs"
    scope -set $currentscope
} elseif {[string match "*.function *" $i]} {
    puts -nonewline "$tabs"
    puts "function($name)"
    printsopes "$currentscope.$name" "      $tabs"
    scope -set $currentscope
} elseif {[string match "*sequential block*" $i]} {
    puts -nonewline "$tabs"
    puts "sequential block ($name)"
    printsopes "$currentscope.$name" "      $tabs"
    scope -set $currentscope
} elseif {[string match "*parallel block*" $i]} {
    puts -nonewline "$tabs"
    puts "named block ($name)"
    printsopes "$currentscope.$name" "      $tabs"
    scope -set $currentscope
}
}
}
```

# D

---

## **Code Examples**

---

This appendix contains the example models that are used in the NC-Verilog simulator and NC-VHDL simulator online help.

## harddrive

```
//File: harddrive.v
//Top module
module harddrive;
reg [3:0] data;
reg clk, clr;
wire [3:0] q;
`define stim #100 data=4'b
event end_first_pass;
hardreg h1 (data,clk, clr, q);
initial
begin
    clr = 1;
    clk = 0;
end
always #50 clk = ~clk;
always @(end_first_pass)
    clr = ~clr;
always @ (posedge clk)
    $strobe("at time %0d clr=%b data=%d q=%d", $time, clr, data, q);
initial
begin
repeat (2)
begin
    data = 4'b0000;
`stim 0001;
`stim 0010;
`stim 0011;
`stim 0100;
`stim 0101;
`stim 0110;
`stim 0111;
`stim 1000;
`stim 1001;
`stim 1010;
`stim 1011;
`stim 1100;
`stim 1101;
`stim 1110;
`stim 1111;
```

## NC-Verilog Simulator Help

### Code Examples

---

```
#200 ->end_first_pass;
end
$finish;
end
endmodule
```

```
// File: hardreg.v
module hardreg (d, clk, clrb, q);
input clk, clrb;
input [3:0] d;
output [3:0] q;
flop f1 (d[0], clk, clrb, q[0],),
         f2 (d[1], clk, clrb, q[1],),
         f3 (d[2], clk, clrb, q[2],),
         f4 (d[3], clk, clrb, q[3],);
endmodule
```

```
// File: flop.v
module flop (data, clock, clear, q, qb);
input data, clock, clear;
output q, qb;
nand #10 nd1 (a, data, clock, clear),
            nd2 (b, ndata, clock),
            nd4 (d, c, b, clear),
            nd5 (e, c, nclock),
            nd6 (f, d, nclock),
            nd8 (qb, q, f, clear);
nand #9  nd3 (c, a, d),
            nd7 (q, e, qb);
not  #10 iv1 (ndata, data),
            iv2 (nclock, clock);
endmodule
```

## harddrive1

```
// File: harddrive1.v
// Top module
module harddrive1;
reg [3:0] data;
reg clk, clr;
wire [3:0] q;
`define stim #100 data=4'b
event end_first_pass;
hardreg h1 (data,clk, clr, q);
initial
begin
    clr = 1;
    clk = 0;
end
always #50 clk = ~clk;
always @(end_first_pass)
    clr = ~clr;
initial
begin
repeat (2)
begin
    data = 4'b0000;
    `stim 0001;
    `stim 0010;
    `stim 0011;
    `stim 0100;
    `stim 0101;
    `stim 0110;
    `stim 0111;
    `stim 1000;
    `stim 1001;
    `stim 1010;
    `stim 1011;
    `stim 1100;
    `stim 1101;
    `stim 1110;
    `stim 1111;
#200 ->end_first_pass;
end
```

## NC-Verilog Simulator Help

### Code Examples

---

```
$finish;  
end  
endmodule  
  
// File: hardreg.v  
module hardreg (d, clk, clrb, q);  
  input clk, clrb;  
  input [3:0] d;  
  output [3:0] q;  
  flop f1 (d[0], clk, clrb, q[0]),  
           f2 (d[1], clk, clrb, q[1]),  
           f3 (d[2], clk, clrb, q[2]),  
           f4 (d[3], clk, clrb, q[3]);  
endmodule  
  
// File: flop.v  
module flop (data, clock, clear, q, qb);  
  input data, clock, clear;  
  output q, qb;  
  nand #10 nd1 (a, data, clock, clear),  
    nd2 (b, ndata, clock),  
    nd4 (d, c, b, clear),  
    nd5 (e, c, nclock),  
    nd6 (f, d, nclock),  
    nd8 (qb, q, f, clear);  
  nand #9 nd3 (c, a, d),  
    nd7 (q, e, qb);  
  not #10 iv1 (ndata, data),  
    iv2 (nclock, clock);  
endmodule
```

## shortdrive

```
// File: shortdrive.v
//Top module
module shortdrive;
    reg [3:0] data;
    reg clk, clr;
    wire [3:0] q;
hardreg h1 (data, clk, clr, q);
initial
begin
repeat (2)
begin
    data = 4'b0;
    #100 data = 4'b1;
    #100 data = 4'b10;
end
$finish;
end
endmodule
```

```
// File: hardreg.v
module hardreg (d, clk, clrb, q);
input clk, clrb;
input [3:0] d;
output [3:0] q;
flop f1 (d[0], clk, clrb, q[0],),
         f2 (d[1], clk, clrb, q[1],),
         f3 (d[2], clk, clrb, q[2],),
         f4 (d[3], clk, clrb, q[3],);
endmodule
```

```
// File: flop.v
module flop (data, clock, clear, q, qb);
input data, clock, clear;
output q, qb;
nand #10 nd1 (a, data, clock, clear),
            nd2 (b, ndata, clock),
            nd4 (d, c, b, clear),
```

## NC-Verilog Simulator Help

### Code Examples

---

```
nd5 (e, c, nclock),
nd6 (f, d, nclock),
nd8 (qb, q, f, clear);
nand #9 nd3 (c, a, d),
      nd7 (q, e, qb);
not #10 iv1 (ndata, data),
        iv2 (nclock, clock);
endmodule
```

## board

```
// File: board.v
module board;
    wire [3:0] count;
    wire clock, f, af;
    m16 counter (count, clock, f, af);
    m555 clock (clock);
    always @(posedge clock)
        $display($time,,,"count=%d, f=%d, af=%d", count, f, af);
endmodule

// File: counter.v
module m16(value, clock, fifteen, altFifteen);
    output [3:0] value;
    output fifteen, altFifteen;
    input clock;
    dEdgeFF a(value[0], clock, ~value[0]),
            b(value[1], clock, value[1] ^ value[0]),
            c(value[2], clock, value[2] ^ &value[1:0]),
            d(value[3], clock, value[3] ^ &value[2:0]);
    assign fifteen = value[0] & value[1] & value[2] & value[3];
    assign altFifteen = &value;
endmodule

// File: clock.v
module m555(clock);
    output clock;
    reg clock;
    initial
        #5 clock = 1;
    always
        #50 clock = ~clock;
endmodule
```

## NC-Verilog Simulator Help

### Code Examples

---

```
// File: ff.v
module dEdgeFF(q, clock, data);
    output q;
    reg q;
    input clock, data;
    initial
        #10 q = 0;
    always
        @(negedge clock)#10 q=data;
endmodule
```

## count

```
// File: count.v
module count;
reg clr, clk;
reg [3:0] data;
wire [3:0] q;
hardreg h1 (data, clk, clr, q);
initial
begin
    assign clr = 0;
    clk = 0;
end
always #50 clk = ~clk;
always @(posedge clk)
    $display("at time %0d clr =%b data=%d q=%d", $time, clr, data, q);
initial
begin
repeat (2)
begin
    data = 4'b0000;
#100 data = 4'b0001;
#100 data = 4'b0010;
#100 data = 4'b0011;
#100 data = 4'b0100;
#100 data = 4'b0101;
#100 data = 4'b0110;
#100 data = 4'b0111;
#100 data = 4'b1000;
#100 data = 4'b1001;
#100 data = 4'b1010;
#100 data = 4'b1011;
#100 data = 4'b1100;
#100 data = 4'b1101;
#100 data = 4'b1110;
#100 data = 4'b1111;
#200;
end
$finish;
end
endmodule
```

## NC-Verilog Simulator Help

### Code Examples

---

```
// File: hardreg.v
module hardreg (d, clk, clrb, q);
input clk, clrb;
input [3:0] d;
output [3:0] q;
flop f1 (d[0], clk, clrb, q[0], ),
         f2 (d[1], clk, clrb, q[1], ),
         f3 (d[2], clk, clrb, q[2], ),
         f4 (d[3], clk, clrb, q[3], );
endmodule

// File: flop.v
module flop (data, clock, clear, q, qb);
input data, clock, clear;
output q, qb;
nand #10 nd1 (a, data, clock, clear),
            nd2 (b, ndata, clock),
            nd4 (d, c, b, clear),
            nd5 (e, c, nclock),
            nd6 (f, d, nclock),
            nd8 (qb, q, f, clear);
nand #9  nd3 (c, a, d),
nd7 (q, e, qb);
not  #10 iv1 (ndata, data),
iv2 (nclock, clock);
endmodule
```

## adder

```
// File: half_adder.v
module half_adder (sum, c_out, a, b);
    input a, b;
    output sum, c_out;
    xor #10 n1 (sum, a, b);
    and #10 n2 (c_out, a, b);
endmodule

// File: full_adder.v
module full_adder (sum,c_out,a,b,c_in);
    input a, b, c_in;
    output sum, c_out;
    half_adder m1 (si, cil, a, b),
                m2 (sum, ci2, si, c_in);
    or m3 (c_out, ci2, cil);
endmodule

// File: two_bit_adder.v
module two_bit_adder (sum,c_out,a,b,c_in);
    input [1:0] a, b;
    input c_in;
    output [1:0] sum;
    output c_out;
    wire [1:0] a, b, sum;
    wire c_out1, c_out2, c_out3;
    full_adder p0 (sum[0], c_out1, a[0], b[0], c_in), p1 (sum[1], c_out, a[1],
                b[1], c_out1);
endmodule

// File: tester.v
module tester;
    reg [1:0] bus_a, bus_b;
    reg c_in;
    wire [1:0] sum;
    wire c_out;
    two_bit_adder under_test (sum, c_out, bus_a, bus_b, c_in);
```

## NC-Verilog Simulator Help

### Code Examples

---

```
initial
begin
    c_in  = 1'b0;
    bus_a = 2'b00;
    bus_b = 2'b01;
#2000 bus_a = 2'b01;
#1000 c_in = 2'b01;
#2000 bus_a = 2'b10;
#1000 c_in = 2'b00;
end
always @({bus_a,bus_b})
#100 // time for output to stabilize
begin
    $display("At time %0d  %b + %b (+%b)= %b\n", $time, bus_a, bus_b, c_in,
             sum);
    if (((bus_a + bus_b + c_in) & 'b11) != sum) $stop;
end
endmodule
```

## register

```
// File: register_test.v
module top;
    wire [7:0] reg_out; //declare vector for register output
    reg [7:0] data;
    reg ena, rst, start, error;
    register rl(reg_out, clk, data, ena, rst);
    nand #10 (clk, clk, start); //clock oscillator
    initial //start the clock oscillator
        begin
            start = 0;
            #10 start = 1;
        end
    initial //apply stimulus to register inputs
        begin
            rst = 0; //should reset register to hex 00
            ena = 1; data = 8'hff; //prepare to load the register
            @(posedge clk); //should NOT load data with rst asserted
            #2 rst = 1; //de-assert reset
            @(posedge clk); //should load data (hex FF)
            @(negedge clk) data = 8'h55; //sync to negedge clock to meet setup spec.
            @(posedge clk); //should load data (hex 55)
            #10 ena = 0; data = 8'hff; //de-assert enable
            @(posedge clk); //should NOT load data with ena deasserted
            #20 $stop;
            $finish;
        end
    initial
        begin
            error=0;
            #20 if(reg_out !== 8'h00)
                begin
                    $display($stime,, "out should be 8'b00000000 but is
                                8'b%b",reg_out);
                    error=1;
                end
            #20 if(reg_out !== 8'hff)
                begin
                    $display($stime,, "out should be 8'b11111111 but is
                                8'b%b",reg_out);
                end
        end
endmodule
```

## NC-Verilog Simulator Help

### Code Examples

---

```
        error=1;
    end
#20 if(reg_out !== 8'h55)
begin
    $display($stime,, "out should be 8'b01010101 but is
                    8'b%b",reg_out);
    error=1;
end
#1 if (!error)
$display("***** TEST PASSED *****");
else
$display("***** TEST FAILED *****");
end
endmodule

// File: register.v
module register(r, clk, data, ena, rst);
    output [7:0] r;
    input [7:0] data;
    input clk, ena, rst;
    wire [7:0] data, r;
    and a1(load, clk, ena);
    dff d0 (r[0], , data[0], load, rst),
    d1 (r[1], , data[1], load, rst),
    d2 (r[2], , data[2], load, rst),
    d3 (r[2], , data[3], load, rst),
    d4 (r[4], , data[4], load, rst),
    d5 (r[5], , data[5], load, rst),
    d6 (r[6], , data[6], load, rst),
    d7 (r[7], , data[7], load, rts);
endmodule
```

## NC-Verilog Simulator Help

### Code Examples

---

```
// File: dff.v
module dff(q, qb, d, clk, rst);
    output q, qb;
    input clk, d, rst;
    nand n1 (cf, dl, cbf);
    nand n2 (cbf, clk, cf, rst);
    nand n3 (dl, d, dbl, rst);
    nand n4 (dbl, dl, clk, cbf);
    nand n5 (q, cbf, qb);
    nand n6 (qb, dbl, q, rst);
endmodule
```

## top

```
// File: ./top.v
module top;
    reg in1, in2;
    wire out1, out2;
    comb_logic block1(in1, in2, out1, out2);
    initial
        begin
            in1 = 1'b0;
            in2 = 1'b0;
        end
    initial
        begin
            #1000 in1 = 1'b1;
            #1000 in2 = 1'b1;
            #1000 in2 = 1'b0;
            #1000 in1 = 1'b0;
            #1000 $stop;
        end
endmodule
```

```
// File: ./libs/comb_logic.v
module comb_logic (in1, in2, out1, out2);
    input in1, in2;
    output out1, out2;
    or2 gate1 (in1, in2, out1);
    and2 gate2 (in1, in2, out2);
endmodule
```

```
// File: ./models/and2.v
module and2 (A, B, X);
    input A, B;
    output X;
    wire X;
    assign X = (A && B);
endmodule
```

## NC-Verilog Simulator Help

### Code Examples

---

```
// File: ./models/or2.v
module or2 (A, B, X);
    input A, B;
    output X;
    wire X;
    assign X = A || B;
endmodule
```

## drink\_machine

```
// File: DRINK.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity CAN_COUNTER is
    port( CLK: in std_logic;
          LOAD: in std_logic;
          CANS: in std_logic_vector(7 downto 0);
          EMPTY: out std_logic;
          DISPENSE: in std_logic;
          RESET: in std_logic
    );
end;

-- Can Counter Architecture
architecture RTL of CAN_COUNTER is
    signal empty_temp: std_logic;
begin
    empty <= empty_temp;
    COUNT_CANS : process(CLK, LOAD, CANS, DISPENSE, RESET)
    variable LEFT : integer range 0 to 255 := 0;
    begin
        if (RESET = '1') then
            LEFT := 200;
            empty_temp <= '1';
        elsif (CLK'event and CLK = '1') then
            if (LOAD = '1' and dispense = '0') then
                LEFT := to_integer(CANS);
                empty_temp <= '0';
            elsif (LOAD = '0' and dispense = '1') then
                LEFT := LEFT - 1;

                if (LEFT = 0) then
                    empty_temp<= '1';
                    assert false report
                    "Out of order. Call for service."
                    severity note;
                end if;
            end if;
        end if;
    end process;
end;
```

## NC-Verilog Simulator Help

### Code Examples

---

```
        end if; --left=0

    end if; --load/dispense

end if; --clk'event
end process;
end RTL;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity COIN_COUNTER is
port( CLK: in std_logic;
      LOAD: in std_logic;
      NICKEL_OUT, DIME_OUT, TWO_DIME_OUT: in std_logic;
      NICKELS, DIMES: in std_logic_vector(7 downto 0);
      EMPTY: out std_logic;
      RESET: in std_logic
 );
end coin_counter;

architecture RTL of COIN_COUNTER is
signal empty_temp : std_logic;
begin
    empty <= empty_temp;
COUNT_COINS : process( CLK, LOAD, NICKEL_OUT, DIME_OUT, TWO_DIME_OUT,
NICKELS, DIMES, RESET )
variable NICKEL_COUNT : integer range 0 to 255 := 0;
variable DIME_COUNT : integer range 0 to 255 := 0;
begin
begin
    if (RESET = '1')then
        NICKEL_COUNT := 0;
        DIME_COUNT := 0;
        empty_temp <= '1';
    elsif (CLK'event and CLK = '0') then
        if (LOAD = '1') then
            NICKEL_COUNT := to_integer(NICKELS);
            DIME_COUNT := to_integer(DIMES);
        elsif (NICKEL_OUT = '1') then
            NICKEL_COUNT := NICKEL_COUNT - 1;
        end if;
    end if;
end process;
end;
```

## NC-Verilog Simulator Help

### Code Examples

---

```
        elsif (DIME_OUT = '1') then
            DIME_COUNT := DIME_COUNT - 1;
        elsif (TWO_DIME_OUT = '1') then
            DIME_COUNT := DIME_COUNT - 2;
        end if;
        if (DIME_COUNT = 0 and NICKEL_COUNT = 0) then
            empty_temp <= '1';
        else
            empty_temp <= '0';
        end if;
    end if;
end process;
end RTL;

library ieee;
use ieee.std_logic_1164.all;
entity DRINK_MACHINE is
    port(NICKEL_IN, DIME_IN, QUARTER_IN, RESET: in std_logic;
          CLK: in std_logic;
          NICKEL_OUT, DIME_OUT, TWO_DIME_OUT, DISPENSE: out std_logic);
end;

-- Drink Machine Architecture
architecture RTL of DRINK_MACHINE is
    type STATE_TYPE is (
        IDLE, FIVE, TEN, FIFTEEN, TWENTY, TWENTY_FIVE, THIRTY,
        THIRTY_FIVE, FORTY, FORTY_FIVE, FIFTY, OWE_NICKEL,
        OWE_DIME, OWE_NICKEL_DIME, OWE_TWO_DIME
    );
    signal CURRENT_STATE: STATE_TYPE;
    signal nickel_out_temp, dime_out_temp, two_dime_out_Temp, dispense_temp : std_logic;
begin
    NICKEL_OUT <= nickel_out_temp;
    DIME_OUT <= dime_out_temp;
    TWO_DIME_OUT <= two_dime_out_Temp;
    DISPENSE <= dispense_temp;
```

## NC-Verilog Simulator Help

### Code Examples

---

```
VENDING : process(CLK, RESET, NICKEL_IN, DIME_IN, QUARTER_IN )
begin
    if (RESET='1') then
        CURRENT_STATE <= IDLE;
        nickel_out_temp <= '0';
        dime_out_temp <= '0';
        dispense_temp <= '0';
    elsif (CLK'event and CLK='1') then
        -- State transitions and output logic
    case CURRENT_STATE is
        when IDLE =>
            nickel_out_temp <= '0';
            dime_out_temp <= '0';
            dispense_temp <= '0';
            if(NICKEL_IN='1') then
                CURRENT_STATE <= FIVE;
            elsif(DIME_IN='1') then
                CURRENT_STATE <= TEN;
            elsif(QUARTER_IN='1') then
                CURRENT_STATE <= TWENTY_FIVE;
            end if;
        when FIVE =>
            if(NICKEL_IN='1') then
                CURRENT_STATE <= TEN;
            elsif(DIME_IN='1') then
                CURRENT_STATE <= FIFTEEN;
            elsif(QUARTER_IN='1') then
                CURRENT_STATE <= THIRTY;
            end if;
        when TEN =>
            if(NICKEL_IN='1') then
                CURRENT_STATE <= FIFTEEN;
            elsif(DIME_IN='1') then
                CURRENT_STATE <= TWENTY;
            elsif(QUARTER_IN='1') then
                CURRENT_STATE <= IDLE;
                dispense_temp <= '1';
            end if;
        when FIFTEEN =>
            if(NICKEL_IN='1') then
                CURRENT_STATE <= TWENTY;
```

## NC-Verilog Simulator Help

### Code Examples

---

```
elsif(DIME_IN='1') then
    CURRENT_STATE <= TWENTY_FIVE;
elsif(QUARTER_IN='1') then
    CURRENT_STATE <= IDLE;
    dispense_temp <= 'Z';
    nickel_out_temp <= '1';
end if;

when TWENTY =>
if(NICKEL_IN='1') then
    CURRENT_STATE <= TWENTY_FIVE;
elsif(DIME_IN='1') then
    CURRENT_STATE <= THIRTY;
elsif(QUARTER_IN='1') then
    CURRENT_STATE <= IDLE;
    dispense_temp <= '1';
    dime_out_temp <= '1';
end if;

when TWENTY_FIVE =>
if(NICKEL_IN='1') then
    CURRENT_STATE <= THIRTY;
elsif(DIME_IN='1') then
    CURRENT_STATE <= THIRTY_FIVE;
    dispense_temp <= '1';
elsif(QUARTER_IN='1') then
    CURRENT_STATE <= FIFTY;
end if;

when THIRTY =>
if(NICKEL_IN='1') then
    CURRENT_STATE <= THIRTY_FIVE;
    dispense_temp <= '1';
elsif(DIME_IN='1') then
    CURRENT_STATE <= FORTY;
elsif(QUARTER_IN='1') then
    CURRENT_STATE <= OWE_NICKEL;
end if;

when THIRTY_FIVE =>
if(NICKEL_IN='1') then
    CURRENT_STATE <= FORTY;
    dispense_temp <= '1';
elsif(DIME_IN='1') then
    CURRENT_STATE <= FORTY_FIVE;
```

## NC-Verilog Simulator Help

### Code Examples

---

```
        elsif(QUARTER_IN='1') then
            CURRENT_STATE <= OWE_DIME;
        end if;
    when FORTY =>
        if(NICKEL_IN='1') then
            CURRENT_STATE <= FORTY_FIVE;
            dispense_temp <= '1';
        elsif(DIME_IN='1') then
            CURRENT_STATE <= FIFTY;
        elsif(QUARTER_IN='1') then
            CURRENT_STATE <= OWE_NICKEL_DIME;
        end if;
    when FORTY_FIVE =>
        if(NICKEL_IN='1') then
            CURRENT_STATE <= FIFTY;
            dispense_temp <= '1';
        elsif(DIME_IN='1') then
            CURRENT_STATE <= OWE_NICKEL;
        elsif(QUARTER_IN='1') then
            CURRENT_STATE <= OWE_TWO_DIME;
        end if;
    when FIFTY =>
        dispense_temp <= '1';
    when OWE_NICKEL =>
        dispense_temp <= '0';
        CURRENT_STATE <= IDLE;
        nickel_out_temp <= '1';
    when OWE_DIME =>
        dispense_temp <= '0';
        CURRENT_STATE <= IDLE;
        dime_out_temp <= '1';
    when OWE_TWO_DIME =>
        dispense_temp <= '1';
        CURRENT_STATE <= IDLE;
        two_dime_out_temp <= '1';
    when OWE_NICKEL_DIME =>
        dispense_temp <= '1';
        CURRENT_STATE <= IDLE;
        dime_out_temp <= '1';
        nickel_out_temp <= '1';
end case;
```

## NC-Verilog Simulator Help

### Code Examples

---

```
    end if;
end process;

end RTL;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY drink_machine_top IS
PORT(
    NICKEL_OUT : out std_logic;
    EMPTY : OUT std_logic;
    EXACT_CHANGE : OUT std_logic;
    TWO_DIME_OUT : out std_logic;
    DIME_OUT : out std_logic;
    DISPENSE : out std_logic;
    CANS : IN std_logic_vector(7 DOWNTO 0);
    CLK : IN std_logic;
    DIMES : IN std_logic_vector(7 DOWNTO 0);
    QUARTER_IN : IN std_logic;
    DIME_IN : IN std_logic;
    LOAD : IN std_logic;
    NICKEL_IN : IN std_logic;
    NICKELS : IN std_logic_vector(7 DOWNTO 0);
    RESET : IN std_logic);
END drink_machine_top;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ARCHITECTURE structure OF drink_machine_top IS
COMPONENT COIN_COUNTER
PORT(
    CLK : IN std_logic;
    LOAD : IN std_logic;
    NICKEL_OUT : in std_logic;
    DIME_OUT : in std_logic;
    TWO_DIME_OUT : in std_logic;
    NICKELS : IN std_logic_vector(7 DOWNTO 0);
    DIMES : IN std_logic_vector(7 DOWNTO 0);
    EMPTY : out std_logic;
```

## NC-Verilog Simulator Help

### Code Examples

---

```
        RESET : in std_logic
    );
END COMPONENT;

COMPONENT CAN_COUNTER
PORT(
    CLK : IN  std_logic;
    LOAD : IN  std_logic;
    CANS : IN  std_logic_vector(7 DOWNTO 0);
    EMPTY : out  std_logic;
    DISPENSE : in  std_logic;
    RESET : in  std_logic
);
END COMPONENT;

COMPONENT DRINK_MACHINE
PORT(
    NICKEL_IN : IN  std_logic;
    DIME_IN : IN  std_logic;
    QUARTER_IN : IN  std_logic;
    RESET : IN  std_logic;
    CLK : IN  std_logic;
    NICKEL_OUT : out  std_logic;
    DIME_OUT : out  std_logic;
    TWO_DIME_OUT : out  std_logic;
    DISPENSE : out  std_logic);
END COMPONENT;

for all : coin_counter use entity work.coin_counter(rtl);
for all : can_counter use entity work.can_counter(rtl);
for all : drink_machine use entity work.drink_machine(rtl);

SIGNAL DISPENSE_tempsig : std_logic;
SIGNAL DIME_OUT_tempsig : std_logic;
SIGNAL TWO_DIME_OUT_tempsig : std_logic;
SIGNAL NICKEL_OUT_tempsig : std_logic;

BEGIN
    DISPENSE <= DISPENSE_tempsig;
    DIME_OUT <= DIME_OUT_tempsig;
    TWO_DIME_OUT <= TWO_DIME_OUT_tempsig;
```

## NC-Verilog Simulator Help

### Code Examples

---

```
NICKEL_OUT <= NICKEL_OUT_tempsig;

COINS : COIN_COUNTER PORT MAP (CLK => CLK, DIMES =>
DIMES, EMPTY => EXACT_CHANGE, NICKEL_OUT =>
NICKEL_OUT_tempsig, LOAD => LOAD, NICKELS =>
NICKELS, DIME_OUT => DIME_OUT_tempsig, TWO_DIME_OUT =>
TWO_DIME_OUT_tempsig, RESET => RESET);

DRINKS : CAN_COUNTER PORT MAP (CANS => CANS,
CLK => CLK, EMPTY => EMPTY, LOAD => LOAD, DISPENSE =>
DISPENSE_tempsig,RESET => RESET);

VENDING : DRINK_MACHINE PORT MAP (QUARTER_IN => QUARTER_IN, CLK =>
CLK, NICKEL_OUT => NICKEL_OUT_tempsig, DIME_IN => DIME_IN,
NICKEL_IN => NICKEL_IN, RESET => RESET, DIME_OUT =>
DIME_OUT_tempsig, DISPENSE => DISPENSE_tempsig, TWO_DIME_OUT =>
TWO_DIME_OUT_tempsig);

END structure;

// File: tb.vhd
-- testbench generated by
-- Program: hdltb
-- on May 5 10:13:16 1998
-- Release Version: hdltb 1.20-p001

-- library declaration
library worklib;
library ieee;
use IEEE.std_logic_1164.all;
-- entity declaration
entity drink_machine_top_test is
end drink_machine_top_test;

-- architecture generated by
-- Program: hdltb
-- on May 5 10:13:16 1998
-- Release Version: hdltb 1.20-p001
-- Copyright: hdltb 1.20-p001: (c) Copyright 1992-1997, Cadence Design Systems,
Inc.
```

## NC-Verilog Simulator Help

### Code Examples

---

```
-- architecture declaration
architecture tbench_1 of drink_machine_top_test is
    signal stoppit : BOOLEAN:= TRUE;

    component drink_machine_top
        port(
            NICKEL_OUT out std_logic ;
            EMPTY out std_logic ;
            EXACT_CHANGE out std_logic ;
            TWO_DIME_OUT out std_logic ;
            DIME_OUT out std_logic ;
            DISPENSE out std_logic ;
            CANS in std_logic_vector( 7 downto 0 ) ;
            CLK in std_logic ;
            DIMES in std_logic_vector( 7 downto 0 ) ;
            QUARTER_IN in std_logic ;
            DIME_IN in std_logic ;
            LOAD in std_logic ;
            NICKEL_IN in std_logic ;
            NICKELS in std_logic_vector( 7 downto 0 ) ;
            RESET in std_logic
        );
    end component ;

    for all : drink_machine_top use entity work.drink_machine_top(structure);

    signal t_NICKEL_OUT std_logic;
    signal t_EMPTY std_logic ;
    signal t_EXACT_CHANGE std_logic ;
    signal t_TWO_DIME_OUT std_logic ;
    signal t_DIME_OUT std_logic ;
    signal t_DISPENSE std_logic ;
    signal t_CANS std_logic_vector( 7 downto 0 ) ;
    signal t_CLK std_logic := '0' ;
    signal t_DIMES std_logic_vector( 7 downto 0 ) ;
    signal t_QUARTER_IN std_logic ;
    signal t_DIME_IN std_logic ;
    signal t_LOAD std_logic ;
    signal t_NICKEL_IN std_logic ;
    signal t_NICKELS std_logic_vector( 7 downto 0 ) ;
    signal t_RESET std_logic ;
```

## NC-Verilog Simulator Help

### Code Examples

---

```
begin

    top : drink_machine_top
    port map (
        NICKEL_OUT => t_NICKEL_OUT ,
        EMPTY => t_EMPTY ,
        EXACT_CHANGE => t_EXACT_CHANGE ,
        TWO_DIME_OUT => t_TWO_DIME_OUT ,
        DIME_OUT => t_DIME_OUT ,
        DISPENSE => t_DISPENSE ,
        CANS => t_CANS ,
        CLK => t_CLK ,
        DIMES => t_DIMES ,
        QUARTER_IN => t_QUARTER_IN ,
        DIME_IN => t_DIME_IN ,
        LOAD => t_LOAD ,
        NICKEL_IN => t_NICKEL_IN ,
        NICKELS => t_NICKELS ,
        RESET => t_RESET
    ) ;

load_nickels : t_NICKELS  <= "11111111";
load_dimes : t_DIMES <= "11111111";
load_cans : t_CANS <= "11111111";
load_action : t_LOAD <= '1' , '0' after 300 ns;

gen_clk : t_clk <= not(t_clk) after 100 ns;

gen_reset : t_reset <= '1','0' after 100 ns;

gen_nickels : t_NICKEL_IN <= '0',
               '1' after 1200 ns,
               '0' after 1300 ns,
               '1' after 3500 ns,
               '0' after 3800 ns,
               '1' after 4600 ns,
               '0' after 4700 ns;

gen_dimes : t_DIME_IN <= '0',
            '1' after 600 ns,
            '0' after 700 ns,
```

## NC-Verilog Simulator Help

### Code Examples

---

```
'1' after 1400 ns,
'0' after 1500 ns,
'1' after 2000 ns,
'0' after 2100 ns,
'1' after 2400 ns,
'0' after 2500 ns,
'1' after 3800 ns,
'0' after 4000 ns;

gen_quarters : t_QUARTER_IN <= '0',
               '1' after 800 ns,
               '0' after 900 ns,
               '1' after 1600 ns,
               '0' after 1700 ns,
               '1' after 2600 ns,
               '0' after 2700 ns,
               '1' after 3000 ns,
               '0' after 3200 ns,
               '1' after 4500 ns,
               '0' after 4600 ns,
               '1' after 4700 ns,
               '0' after 4800 ns,
               '1' after 5000 ns,
               '0' after 5100 ns,
               '1' after 5200 ns,
               '0' after 5300 ns;

stoppit <= false after 6000 ns;
assert stoppit report "Completed simulation" severity failure;
end tbench_1;
```

## NC-Verilog Simulator Help

### Code Examples

---

```
// File: conf_rtl.vhd
-- Configurations for top-level unit WORKLIB.DRINK_MACHINE_TOP_TEST:TBENCH_1
-- Configuration Model: Flat
-- No priority list of architectures specified
--
-- Libraries
library WORKLIB;
library IEEE;

configuration cfg_rtl of DRINK_MACHINE_TOP_TEST is
    for TBENCH_1
        for OTHERS: DRINK_MACHINE_TOP use entity WORKLIB.DRINK_MACHINE_TOP(structure);
        for structure
            for OTHERS: COIN_COUNTER use entity WORKLIB.COIN_COUNTER(rtl);
                for rtl
                    end for;
                end for;
            end for;
            for OTHERS: CAN_COUNTER use entity WORKLIB.CAN_COUNTER(rtl);
                for rtl
                    end for;
                end for;
            end for;
            for OTHERS: DRINK_MACHINE use entity WORKLIB.DRINK_MACHINE(rtl);
                for rtl
                    end for;
                end for;
            end for;
        end for;
    end configuration cfg_rtl;
```

## drivers.vhd

```
library ieee;
use ieee.std_logic_1164.all;
entity e is
end e;
architecture a of e is
    signal s: std_logic;
    function bit_to_std (x: bit) return std_logic is
    begin
        return '0';
    end bit_to_std;
    function std_to_bit (x: std_logic) return bit is
    begin
        return '1';
    end std_to_bit;
    begin
        s <= '0' after 1 ns;
        GATE: block
            port (q: inout bit);
            port map (bit_to_std (q) => std_to_bit (s));
            begin
                p: process (q)
                    begin
                        q <= not q;
                    end process;
            end block;
    end;
```

## Verilog-VHDL-Verilog Sandwich

```
// File: sub.v
module vlog(io, c0);
    inout io;
    input c0;

    reg r_io;
    wire io = r_io;

    always @(io or c0)
        $display("%t %m ctrl=%v io=%v", $time, c0, io);

    always @(c0)
        begin
            if (c0 == 1'b1)
                begin
                    $display("drive X\n"); r_io = 1'bx;
                    #10 $display("drive 0\n"); r_io = 1'b0;
                    #10 $display("drive 1\n"); r_io = 1'b1;
                    #10 $display("drive Z\n"); r_io = 1'bz;
                    #10 $display("drive X\n"); r_io = 1'bx;
                    #10 $display("drive 0\n"); r_io = 1'b0;
                    #10 $display("drive 1\n"); r_io = 1'b1;
                    #10 $display("drive Z\n"); r_io = 1'bz;
                end
            else
                r_io = 1'bz;
        end
endmodule
```

The following version of the file `middle.vhd` contains a component declaration for importing the Verilog module `vlog`, described in `sub.v`. No shell is required for importing the Verilog module.

```
-- File: middle.vhd
library ieee;
use ieee.std_logic_1164.all;
library worklib;

entity middle is
    port (io :inout std_logic;
          vctrl : in std_logic_vector(1 downto 0));
```

## NC-Verilog Simulator Help

### Code Examples

---

```
end middle;

architecture A of middle is

    component vlog
        port (
            io : inout std_logic;
            c0 : in std_logic
        );
    end component;

    signal ctrl : std_ulogic;

begin
    v1: vlog
        port map(
            io,
            vctrl(1)
        );

    ctrl <= vctrl(0);

    process
    begin
        wait on ctrl;
        if (ctrl = '1') then
            for val in STD_ULOGIC'LEFT to STD_ULOGIC'RIGHT loop
                io <= val;
                wait for 10 ns;
            end loop;
        end if;
        io <= 'Z';
    end process;

    process (io)
    begin
        assert FALSE
        report "ctrl = " & STD_ULOGIC'IMAGE(ctrl) & ":middle:io = " &
            STD_LOGIC'IMAGE(io)
            severity NOTE;
    end process;
end A;
```

## NC-Verilog Simulator Help

### Code Examples

---

The following version of the file `middle.vhd` references the Verilog module in an entity aspect of a component configuration. The following configuration specification specifies the binding:

```
for all:vlog use entity worklib.vlog(module);
-- File: middle.vhd
library ieee;
use ieee.std_logic_1164.all;
library worklib;

entity middle is
    port (io : inout std_logic;
          vctrl : in std_logic_vector(1 downto 0));
end middle;

architecture A of middle is

component vlog
    port (
        io : inout std_logic;
        c0 : in std_logic
    );
end component;

for all:vlog use entity worklib.vlog(module);

signal ctrl : std_ulogic;

begin
    v1: vlog
        port map(
            io,
            vctrl(1)
        );

    ctrl <= vctrl(0);

    process
    begin
        wait on ctrl;
        if (ctrl = '1') then
            for val in STD_ULOGIC'LEFT to STD_ULOGIC'RIGHT loop
                io <= val;
            end loop;
        end if;
    end process;
end architecture;
```

## NC-Verilog Simulator Help

### Code Examples

---

```
        wait for 10 ns;
    end loop;
end if;
io <= 'Z';
end process;

process (io)
begin
    assert FALSE
    report "ctrl = " & STD_ULOGIC'IMAGE(ctrl) & ":middle:io = " &
           STD_LOGIC'IMAGE(io)
        severity NOTE;
end process;
end A;
```

The following file contains the top-level Verilog module.

```
// File: top.v
module top;
reg [4:0] vctrl;
reg r_io;
wire c0 = vctrl[4];
wire io = r_io;

middle m10 (io, vctrl[1:0]);
middle m32 (io, vctrl[3:2]);

always @(io or c0)
$display("%t %m ctrl=%v io=%v", $time, c0, io);

always @(c0)
begin
if (c0 == 1'b1)
begin
$display("drive X\n"); r_io = 1'bx;
#10 $display("drive 0\n"); r_io = 1'b0;
#10 $display("drive 1\n"); r_io = 1'b1;
#10 $display("drive Z\n"); r_io = 1'bz;
#10 $display("drive X\n"); r_io = 1'bx;
#10 $display("drive 0\n"); r_io = 1'b0;
#10 $display("drive 1\n"); r_io = 1'b1;
#10 $display("drive Z\n"); r_io = 1'bz;
end
end
```

## NC-Verilog Simulator Help

### Code Examples

---

```
    else
        r_io = 1'bz;
    end

initial
begin
    vctrl = 5'b0000;
#200 vctrl = 5'b00001;
#200 vctrl = 5'b00010;
#200 vctrl = 5'b00100;
#200 vctrl = 5'b01000;
#200 vctrl = 5'b10000;
#200 vctrl = 5'b01010;
#100 vctrl = 5'b00000;
#200 vctrl = 5'b00101;
#100 vctrl = 5'b00000;
#200 vctrl = 5'b01001;
end

endmodule
```

# Glossary

---

## A

**alias**

Shorthand for a command or series of commands

## B

**binding**

The operation of locating a module or udp that is instantiated in another (higher-level) module.

**breakpoint**

A trigger that stops the simulation when a specified condition is true. You can set breakpoints on conditions, objects, times, or lines of source code.

## C

**cell**

An object stored in a library with a unique name. In the Cadence library structure, cells are represented by a directory under the library directory.

**cds.lib**

A file that maps logical library names to physical directory paths.

**CLA**

Cadence Library Access. A procedural interface that parses the *cds.lib* file and provides a procedural interface to the data contained in it.

**COD**

Code File. The file format used to store platform-specific machine code generated by the native code compiler.

**Command File**

A file containing simulator commands used as input to a simulation run.

**D**

**database**

A file containing the logic states of selected signals from a simulation run.

The NC-Verilog simulator supports the Simulation History Manager (SHM) and the Value Change Dump (VCD) database formats.

**delta cycle count**

At any given simulation time, values of nets are first updated and then behaviors that are sensitive to those nets are executed. This two step process may be repeated any number of times because of zero-delays. The delta cycle count represents the number of times the process is repeated for the given simulation time.

**design library**

A collection of cells that describe a single design.

**design unit**

A single cell in a design library. Also referred to as a design module.

**E**

**elaborating**

The process of constructing a design hierarchy based on the instantiation and configuration information in the design. The elaborator creates a simulation snapshot from the design hierarchy.

**F**

**Form**

A dialog box that prompts you for information before a command executes. Forms contain fields and buttons that you use to set the values for a command.

**G**

**Graphical user interface**

The main simulation control window that is displayed when you invoke the simulator. The GUI consists of three windows: the SimControl window, the Navigator window, and the Watch Objects window.

**H**

**HDL design unit**

Any Verilog or VHDL object compilable on its own. For Verilog HDL, this includes module, macromodule, UDP, and config design units. For VHDL, this includes entity, architecture, package, package body, and configuration design units.

**hdl.var**

An ASCII file that contains the setting for the WORK variable, which specifies which library is the working library where NC-Verilog stores HDL design units and derived data, configuration variables, which determine how your design environment is configured, and statements that specify the locations of support files and invocation scripts.

**I**

**I/O Region**

The region on the SimControl graphical user interface that displays the commands that are being executed and simulator output.

**L**

**L.C:V**

Shorthand notation for Library.Cell:View

**LD\_LIBRARY\_PATH**

An environment variable used to define dynamic libraries on Sun platforms. For HP platforms, use the environment variable SHLIB\_PATH.

**library**

A container for cells. In the Cadence library structure, libraries are directories on the file system. The directories are referenced by the logical name of the library.

**M**

**message region**

The display area at the bottom of the SimControl window that displays informative messages from the user interface.

## NC-Verilog Simulator Help

### Glossary

---

#### N

**ncelab**

Cadence's native code elaborator for the NC-Verilog and NC-VHDL simulators.

**ncsim**

Cadence's native code simulator.

**NC-Sim mixed language simulator**

Cadence's native-compiled code mixed-language simulator. NC-Sim includes all capabilities of the NC-Verilog and NC-VHDL simulators.

**NC-Verilog simulator**

Cadence's native-compiled code Verilog compiler and simulator.

**NC-VHDL simulator**

Cadence's native-compiled code VHDL compiler and simulator.

**ncvhdl**

The NC-VHDL compiler.

**ncvlog**

The NC-Verilog compiler.

#### P

**Packing**

Converting a library stored as L.C:V data to a structure that contains the entire library in a more compact and efficient format.

#### R

**reference library**

A collection of cells that describe components used in many designs.

**S**

**setup.loc**

An ASCII file that specifies the search rules used for finding library definition (`cds.lib`) and configuration (`hdl.var`) files. The `setup.loc` file contains a list of directories to search. The order of the directories determines the search order.

**SHLIB\_PATH**

An environment variable used to define dynamic libraries on HP platforms. For Sun platforms, use the environment variable `LD_LIBRARY_PATH`.

**SIG**

Signature File. The file format used to represent unique instantiation signatures of Verilog modules.

**SSS**

Simulation snapshot file. The file format used to store the state of an elaborated design, including simulation run-time data.

**T**

**Tcl**

The Tool Command Language (Tcl) used to control the execution of a simulation. Cadence has extended the functionality of the Tcl command interpreter so that it understands a number of new commands and some new syntax.

**V**

**VPI**

Verilog Programming Interface. A procedural interface that lets programs access the contents of VST files. This conforms to the IEEE-1364 Verilog Standard. This interface also is known as PLI 2.0.

**VST**

Verilog Syntax Tree. The file format produced by the Verilog compiler to represent the Verilog language in an intermediate form. Because the files are system generated, they are generally not a user concern.

## **NC-Verilog Simulator Help**

### Glossary

---

#### **W**

#### **WORK**

An environment variable that defines the library where compiled design units are to be saved.

---

# Index

---

## Symbols

! command [557](#)  
 \$dumpall [461](#)  
 \$dumpfile [460](#)  
 \$dumpflush [461](#)  
 \$dumplimit [461](#)  
 \$dumpoff [461](#)  
 \$dumpon [461](#)  
 \$dumpports [463](#)  
 \$dumpports\_close [465](#)  
 \$dumpvars [460](#)  
 \$hold [706](#)  
 \$nochange [722](#)  
 \$omiCommand system task [965](#)  
 \$period [713](#)  
 \$readmemb [1061](#)  
 \$readmemh [1061](#)  
 \$recordabort [446](#)  
 \$recordclose [446](#)  
 \$recordfile [446](#)  
 \$recordoff [446](#)  
 \$recordon [446](#)  
 \$recordsetup [446](#)  
 \$recordvars [446](#)  
 \$recovery [716](#)  
 \$recrem [719](#)  
 \$removal [717](#)  
 \$sdf\_annotation [803](#)  
     and ncverilog [85](#)  
     examples of [805](#)  
     requirements for automatic  
         annotation [811](#)  
 \$setup [705](#)  
 \$setuphold [707](#)  
 \$shm\_close [440](#)  
 \$shm\_open [440](#)  
 \$shm\_probe [440](#)  
 \$signalscan [446](#)  
 \$skew [715](#)  
 \$UDTF keyword  
     in access file [259](#)  
 \$width [711](#)  
 +all [56](#)  
 +cdslib+ [56](#)  
 +checkargs [56](#)

+compile [56](#)  
 +debug  
     for ncprep [869](#)  
     for ncverilog [56](#)  
 +elaborate [57](#)  
 +expand  
     for ncverilog [57](#)  
 +hdlvar+ [57](#)  
 +import  
     for ncverilog [57](#)  
 +linedebug  
     for ncprep [869](#)  
 +mixedlang  
     for ncverilog [57](#)  
 +name+ [58](#)  
 +ncelabargs+ [58](#)  
 +ncelabexe+ [59](#)  
 +ncerror+  
     for ncprep [869](#)  
     for ncverilog [59](#)  
 +ncfatal+  
     for ncprep [870](#)  
     for ncverilog [60](#)  
 +nclibdirname+ [60](#)  
     for ncprep [870](#)  
 +ncls\_all [61](#)  
 +ncls\_dependents [61](#)  
 +ncls\_snapshots [62](#)  
 +ncls\_source [62](#)  
 +ncsimargs+ [62](#)  
 +ncsimexe+ [63](#)  
 +ncuid+ [63](#)  
 +ncversion  
     for ncverilog [63](#)  
 +ncvlogargs+ [63](#)  
 +noautosdf  
     for ncverilog [64](#)  
 +noupdate [64](#)  
 +overwrite  
     for ncprep [871](#)  
 +ppe [64](#)  
 +redirect+  
     for ncprep [871](#)  
 +sdf\_orig\_dir  
     for ncverilog [65](#)  
 +work+ [65](#)

'noview [173](#)  
 'noworklib [173](#)  
 'undefineall [696](#)  
 'uselib [233](#)  
 'view [173](#)  
 'worklib [173](#)  
 'define [163](#)  
 'ifdef-'endif [163](#)

## A

ABSOLUTE keyword [1007](#)  
 Access  
     to simulation objects [248](#)  
     in ncverilog [51](#)  
 -access [188](#)  
 Access file  
     including [253](#)  
     keywords [256](#)  
     specifying with -afile [189](#)  
     writing [254](#)  
 Adding values to existing delays [1008](#)  
 -addonly  
     for ncpack [862](#)  
 Affirma SimVision analysis  
     environment [671](#)  
 -afile [189](#)  
 alias command [498](#)  
     examples [499](#)  
     modifiers [498](#)  
         -set [498](#)  
         -unset [498](#)  
     syntax [498](#)  
     using to set and unset aliases [491](#)  
 Aliases  
     creating [491](#)  
     displaying information about [491](#)  
     removing [491](#)  
 -all  
     for nchelp [843](#)  
     for ncls [853](#)  
     for ncshell [914](#)  
 all keyword [554](#)  
 -analopts  
     for ncshell [387](#)  
 -analyze  
     for ncshell [387](#)  
 -anno\_simtime [189](#)  
 Annotation  
     PLI/VPI

    at simulation time [189](#)  
     SDF [789](#)  
         file syntax [994](#)  
 -append\_log [400](#)  
     for ncelpab [190](#)  
     for ncexport [835](#)  
     for ncls [853](#)  
     for ncpack [862](#)  
     for ncrm [902](#)  
     for ncsdfc [908](#)  
     for ncshell [915](#)  
     for ncsim [301](#)  
     for ncupdate [929](#)  
     for ncvlog [141](#)  
 -architecture  
     for ncls [853](#)  
 Arrays of instances [91](#)  
 ARRIVAL keyword [1045](#)  
 Assert messages  
     printing extended messages [303](#)  
     suppressing [486](#)  
 assert\_1164\_warnings variable [482](#)  
 assert\_report\_level variable [482](#)  
 assert\_stop\_level variable [482](#)  
 ASSIGN statement  
     in cds.lib file [112](#)  
 Assigning delays  
     to module paths [771](#)  
 -ast  
     for ncsuffix [925](#)  
 attribute command [500](#)  
     examples [501](#)  
     syntax [500](#)  
 Attributes  
     assigning to libraries [112](#)  
     enabling with attribute command [500](#)  
     TMP [112](#)  
 autoscope variable [482](#)

## B

-backward  
     for ncshell [915](#)  
 BASENAME keyword  
     in access file [257](#)  
     in timing access file [268](#)  
 -batch [301](#)  
 Binding [233](#)  
     forcing an explicit binding [233](#)  
 -binding [190](#)

---

- body
  - for ncls [853](#)
- Breakpoints
  - condition [427](#)
  - deleting [433](#)
  - delta [431](#)
  - disabling [433](#)
  - displaying [433](#)
  - enabling [433](#)
  - line [428](#)
  - object [429](#)
  - process [431](#)
  - subprogram [432](#)
  - time [430](#)
- Bus contention
  - check command [508](#)
- Bus contention detection [438](#)
- Bus float
  - check command [508](#)
- Bus float detection [438](#)
- C**
- call command [504](#)
  - examples [507](#)
  - options [506](#)
    - predefined [506](#)
    - systf [506](#)
  - syntax [504](#)
- Cancelled schedules
  - and pulse filtering [288](#)
- cds.lib file [110](#)
  - displaying contents of [116](#)
  - example [114](#)
  - statements [111](#)
    - ASSIGN [112](#)
    - DEFINE [111](#)
    - INCLUDE [111](#)
    - SOFTINCLUDE [112](#)
    - UNDEFINE [111](#)
  - syntax rules [113](#)
- cdslib
  - for ncelab [190](#)
  - for ncexport [835](#)
  - for nchelp [843](#)
  - for ncls [853](#)
  - for ncpack [862](#)
  - for ncrm [902](#)
  - for ncsdfc [908](#)
  - for ncsim [302](#)
- Cell
  - for ncupdate [929](#)
  - for ncvlog [141](#)
- Cell
  - definition of [109](#)
- CELL keyword [1003](#)
- CELLINST keyword
  - in access file [258](#)
  - in timing access file [269](#)
- CELLLIB keyword
  - in access file [258](#)
  - in timing access file [269](#)
- CELLTYPE keyword [1004](#)
- CFC
  - loading applications with -loadcfc [306](#)
- check command [508](#)
  - examples [512](#)
  - modifiers [509](#)
    - delete [510](#)
    - disable [510](#)
    - enable [510](#)
    - show [511](#)
  - options [509](#)
    - contention [509](#)
    - delay [509](#)
    - float [510](#)
    - name [511](#)
  - syntax [508](#)
- checktasks [143](#)
- clean variable [483](#)
- cod
  - for ncsuffix [926](#)
- code
  - for ncls [853](#)
- Code coverage
  - coverage command [516](#)
  - enabling [194](#)
- Coding style guidelines [674](#)
- command
  - for ncls [853](#)
- Command file
  - SDF
    - COMPILED\_SDF\_FILE [791](#)
    - CONFIG\_FILE [792](#)
    - examples [793](#)
    - LOG\_FILE [792](#)
    - MTM\_CONTROL [792](#)
    - SCALE\_FACTORS [793](#)
    - SCALE\_TYPE [793](#)
    - SCOPE [791](#)
    - specifying [795](#)
  - Command files

## NC-Verilog Simulator Help

---

executing with -input [335](#)  
executing with the source  
command [335](#)  
Command syntax conventions [496](#)

Commands  
! [557](#)  
alias [498](#)  
attribute [500](#)  
call [504](#)  
check [508](#)  
coverage [516](#)  
database [517](#)  
deposit [528](#)  
describe [532](#)  
drivers [537](#)  
exec [494](#)  
executing UNIX [494](#)  
exit  
[548](#)  
finish [549](#)  
fmibkpt [550](#)  
force [551](#)  
getting help on [44](#)  
help [554](#)  
history [557](#)  
input [560](#)  
memory [563](#)  
omi [568](#)  
probe [571](#)  
process [587](#)  
release [592](#)  
reset [594](#)  
restart [595](#)  
run [598](#)  
save [602](#)  
scope [608](#)  
set [481](#)  
source [616](#)  
stack [618](#)  
status [623](#)  
stop [624](#)  
strobe [644](#)  
task [651](#)  
time [655](#)  
using wildcards in Tcl [495](#)  
value [658](#)  
version [663](#)  
where [664](#)

-comp  
for ncshell [388, 915](#)

Comparescan [477](#)

-compile [908](#)  
for ncelab [191](#)

COMPILED\_SDF\_FILE [791](#)

Compiler directives  
'noview [173](#)  
'nowork [173](#)  
'uselib [233](#)  
'view [173](#)  
'worklib [173](#)

Compiling  
checking for non-standard system  
tasks [143](#)  
conditional compilation [163](#)  
controlling compilation into  
Lib.Cell:View [165](#)  
defining macros [178](#)

Compiling a Verilog-XL design [393](#)

Compiling Verilog source files [137](#)

COND keyword [1012, 1026](#)

CONDELSSE keyword [1013](#)

-condition  
for strobe [645](#)

Condition breakpoints [427](#)

Conditional compilation [163](#)

Conditioned events  
in timing checks [726](#)

-conffile  
for ncelab [191](#)

-confflat  
for ncelab [194](#)

CONFIG\_FILE [792](#)

-configuration  
for ncls [854](#)

Configuration file [184, 813](#)  
automatically compiling [191](#)  
automatically generating with  
ncelab [184](#)  
example [814](#)  
examples of generating [228](#)  
IGNORE [815](#)  
INTERCONNECT\_DELAY [816](#)  
INTERCONNECT\_MIPD [817](#)  
MAP\_INNER [821](#)  
MODULE [820](#)  
MTM [817](#)  
SCALE\_FACTORS [817](#)  
SCALE\_TYPE [818](#)  
syntax [813](#)  
TURNOFF\_DELAY [819](#)

-confname  
for ncelab [194](#)

---

- context
  - for ncexport [835](#)
- Control Menu
  - Run [325](#)
- Controlling compilation into
  - Lib.Cell:View [165](#)
- Conventions for command syntax [496](#)
- Corruption
  - of library database [39](#)
- coverage
  - for ncelab [194](#)
- coverage command [516](#)
- cputime
  - for ncsdfc [908](#)
- Cycle View [671](#)

## D

- database
  - for ncpack [862](#)
- database command [517](#)
  - examples [525](#)
  - modifiers [520](#)
    - close [525](#)
    - disable [524](#)
    - enable [525](#)
    - open [520](#)
    - setdefault [524](#)
    - show [524](#)
  - options [520](#)
    - compress [520](#)
    - default [521](#)
    - evcd [521](#)
    - event [521](#)
    - into [522](#)
    - maxsize [522](#)
    - shm [523](#)
    - timescale [523](#)
    - vcf [523](#)
  - syntax [519](#)
- Databases
  - changing the default [524](#)
  - closing [420](#)
  - comparing with Comparescan [477](#)
  - compressing an SHM [520](#)
  - disabling [419](#)
  - displaying information about [419](#)
  - dumping all value changes to [521](#)
  - enabling [419](#)
  - limiting the size of [522](#)

- opening [418](#)
- using \$recordvars [446](#)

- DATE keyword [1002](#)
- Deassigning attributes from libraries [112](#)

- Debug scope
  - setting [424](#)
- Debugging
  - comparing databases with
    - Comparescan [477](#)
  - disabling, enabling, deleting, and displaying breakpoints [433](#)
  - displaying information about objects [437](#)
  - displaying waveforms with SimVision
    - Waveform Viewer [440](#)
  - searching for a text string [489](#)
  - setting a condition breakpoint [427](#)
  - setting a delta breakpoint [431](#)
  - setting a line breakpoint [428](#)
  - setting a process breakpoint [431](#)
  - setting a subprogram breakpoint [432](#)
  - setting a time breakpoint [430](#)
  - setting an object breakpoint [429](#)
  - stepping through lines of code [434](#)
- decompile [908](#)

- DEFAULT keyword
  - in access file [256](#)
  - in timing access file [268](#)
- Default radix [480](#)
- define [143](#)

- DEFINE statement
  - in cds.lib file [111](#)
  - in hdl.var file [120](#)
- Defining libraries
  - in cds.lib file [110](#)
- Defining variables
  - in hdl.var file [120](#)

- DELAY keyword [1006](#)
- delay\_mode [195](#)

- DELAYFILE keyword [1001](#)

- Delays
  - adding values to existing delays [1008](#)
  - device [1021](#)
  - distributed [755](#)
  - for a complete net [1018](#)
  - replacing values in existing delays [1007](#)
  - state-dependent path [763](#)

- Delta breakpoints [431](#)
- DEPARTURE keyword [1046](#)

- dependents

for ncls [854](#)  
 deposit command [528](#)  
     examples [530](#)  
     options [529](#)  
         -absolute [529](#)  
         -after [529](#)  
         -inertial [529](#)  
         -relative [529](#)  
         -transport [529](#)  
     syntax [529](#)  
 Depositing values [436](#)  
 describe command [532](#)  
     examples [533](#)  
     syntax [532](#)  
 Design changes  
     updating [332](#)  
 DESIGN keyword [1002](#)  
 Design libraries  
     example structure [133](#)  
     mapping to multiple directories [114](#)  
 Desktop simulator [28](#)  
 Device delay [1021](#)  
 DEVICE keyword [1021](#)  
 DIFF keyword [1044](#)  
 Directories  
     binding multiple to one library [114](#)  
 -directory  
     for ncexport [835](#)  
 -disable\_enht [196](#)  
 display\_unit variable [483](#)  
 Distributed delays  
     and module path delays [784](#)  
 DIVIDER keyword [1002](#)  
 Documents  
     printing [43](#)  
     searching [43](#)  
 drivers command [537](#)  
     examples [542](#)  
     options [537](#)  
         -effective [537](#)  
         -future [537](#)  
         -novalue [538](#)  
         -verbose [538](#)  
     report format of [538](#)  
     syntax [537](#)  
 -dump [566](#)  
 dumports  
     See \$dumports [463](#)  
 dumports\_close  
     See \$dumports\_close [465](#)

## E

Edge-control specifiers [722](#)  
 Edge-sensitive module paths [761](#)  
 Editing source files [488](#)  
 Elaborating  
     overview of [181](#)  
     setting module path pulse controls [280](#)  
 -end  
     for memory command [566](#)  
 -entity  
     for ncls [854](#)  
 Environment  
     customizing [489](#)  
     saving and restoring [490](#)  
 -epulse\_neg [196](#)  
 -epulse\_no\_msg [302](#)  
 -epulse\_ondetect [196](#)  
 -epulse\_onevent [197](#)  
 Equality operators [984](#)  
 -errormax [400](#)  
     for ncslab [197](#)  
     for ncsexport [835](#)  
     for ncspack [862](#)  
     for ncshell [915](#)  
     for ncsim [302](#)  
     for ncupdate [929](#)  
     for ncvlog [143](#)  
 Escaped names  
     mapping to file system names [113](#)  
 EVCD databases [463](#)  
     closing with \$dumports\_close [465](#)  
     generating with \$dumports [463](#)  
     syntax and format of [466](#)  
 -exclfile  
     for ncupdate [929](#)  
 -exclude  
     for ncsexport [835](#)  
     for ncupdate [930](#)  
 exec command [494](#)  
 -exit [303](#)  
 exit command [548](#)  
     syntax [548](#)  
 Explicit TMP libraries [114](#)  
 -extassertmsg  
     for ncsim [303](#)  
 -extend\_tcheck\_data\_limit [197](#)  
 -extend\_tcheck\_reference\_limit [198](#)  
 Extended value change dump  
     See EVCD databases [463](#)

## F

### -file

- for ncelab [199](#)
- for nccls [854](#)
- for ncshell [915](#)
- for ncsim [303](#)
- for ncvlog [144](#)

File locking [39](#)

### File Menu

- Commands
- Source [335](#)
- Exit [338](#)
- Find
- Text [489](#)

finish command [549](#)

- examples [549](#)

- syntax [549](#)

### FMI

- loading applications with -loadfmi [307](#)

fmibkpt command [550](#)

- modifiers [550](#)

- disable [550](#)

- enable [550](#)

- show [550](#)

- syntax [550](#)

### -fmient

- for ncshell [916](#)

### -fmilib

- for ncshell [916](#)

### -force

- for ncrm [902](#)

- for ncupdate [930](#)

force command [551](#)

- examples [552](#)

- syntax [552](#)

Foreign attribute [919](#)

Formatting of time values [318](#)

### Full connection

- in module path delay [768](#)

### Functions

- calling from the command line [504](#)

- for type conversion [991](#)

### Function-valued attributes

- enabling [500](#)

## G

-genafile [200](#)

-generic [200](#)  
    for ncshell [388](#)

Getting help  
    on NC tools [43](#)

GLOBALPATHPULSE keyword [1024](#)

-gui [304](#)

## H

### -h

- for ncprep [869](#)

hdl.var file [118](#)

- displaying contents of [129](#)

- example [129](#)

- statements [120](#)

- DEFINE [120](#)

- INCLUDE [120](#)

- SOFTINCLUDE [121](#)

- UNDEFINE [120](#)

- syntax rules [127](#)

- variables

- LIB\_MAP [121](#)

- NCELABOPTS [122](#)

- NCHELP\_DIR [122](#)

- NCSDFCOPTS [123](#)

- NCSIMOPTS [123](#)

- NCSIMRC [123](#)

- NCUPDATEOPTS [123](#)

- NCUSE5X [124](#)

- NCVERILOGOPTS [124](#)

- NCVHDLOPTS [124](#)

- NCVLOGOPTS [125](#)

- SRC\_ROOT [125](#)

- VERILOG\_SUFFIX [125](#)

- VHDL\_SUFFIX [125](#)

- VIEW [126](#)

- VIEW\_MAP [126](#)

- VXLOPTS [126](#)

- WORK [127](#)

### hdl.var variables

- for ncelab [232](#)

- for ncsim [321](#)

- for ncvlog [160](#)

### -hdlvar

- for ncelab [201](#)

- for ncexport [836](#)

- for nchelp [843](#)

- for nccls [854](#)

- for ncpack [863](#)

- for ncrm [902](#)

for ncsdfc [908](#)  
for ncsim [304](#)  
for ncupdate [930](#)  
for ncvlog [146](#)

Header  
of SDF file [1001](#)

Help  
invoking [41](#)  
on ncvlog, ncelab, ncsim messages [44](#)  
on simulator commands [44](#)  
online [41](#)  
    invoking [42](#)  
using [41](#)

-help [43, 400](#)  
for ncelab [202](#)  
for ncexport [836](#)  
for nchelp [844](#)  
for ncls [855](#)  
for ncpack [863](#)  
for ncrm [902](#)  
for ncsdfc [908](#)  
for ncshell [916](#)  
for ncsim [305](#)  
for ncsuffix [926](#)  
for ncupdate [930](#)  
for ncverilog [57](#)  
for ncvlog [146](#)

help command [554](#)  
examples [555](#)  
options [555](#)  
    -brief [555](#)  
    -functions [555](#)  
    -variables [555](#)  
syntax [554](#)

Hierarchy [424](#)  
traversing with scope command [424](#)

History  
of commands [491](#)

history command [557](#)  
examples [558](#)  
options [557](#)  
    keep [558](#)  
    redo [557](#)  
    substitute [558](#)  
syntax [557](#)  
using to display commands [491](#)

HOLD keyword [1028](#)

|

-ieee  
    for ncupdate [930](#)  
IEEE-1364  
    compliance with [90](#)

-ieee1364  
    for ncelab [202](#)  
    for ncvlog [146](#)

IGNORE [815](#)

Implicit TMP libraries [115](#)

-import  
    for ncshell [916](#)  
    for Verilog or VHDL import [389](#)

INCA [26](#)

-incdir [147](#)

-include  
    for ncexport [836](#)

Include files  
specifying a directory to search [147](#)

INCLUDE keyword [1005](#)  
in access file [259](#)  
in timing access file [269](#)

INCLUDE statement  
in cds.lib file [111](#)  
in hdl.var file [120](#)

Including files  
in cds.lib file [111](#)  
in hdl.var file [120](#)

INCREMENT keyword [1008](#)

-input [305](#)

input command [560](#)  
examples [561](#)  
syntax [561](#)

Input files  
executing with -input [335](#)  
executing with the source  
    command [335](#)  
    sourcing [616](#)

INSTANCE keyword [1004](#)

Instances  
arrays of [91](#)

Interconnect delays  
setting pulse controls [280](#)

INTERCONNECT keyword [1016](#)

INTERCONNECT\_DELAY [816](#)

INTERCONNECT\_MIPD [817](#)

Interleaved Native Compiled Code  
Architecture (INCA) [26](#)

-intermod\_path [202](#)

## NC-Verilog Simulator Help

---

-into  
  for ncshell [916](#)  
  for Verilog or VHDL import [389](#)  
IOPATH keyword [1009](#)

## K

-keepvalue [593](#)  
-keyfile [305](#)

## L

-l  
  for ncprep [869](#)  
Language rules  
  VHDL  
    relaxing [217](#)  
Launch tool (NCLaunch) [48](#)  
-lexpragma  
  for ncvlog [149](#)  
LIB\_MAP variable [121](#)  
-libcell [149](#)

Libraries  
  assigning attributes to [112](#)  
  defining in cds.lib file [110](#)  
  example structure [133](#)  
  undefining [111](#)

Library  
  binding to multiple directories [114](#)  
  definition of [109](#)

-library  
  for ncls [855](#)  
  for ncrm [902](#)  
  for ncupdate [930](#)  
Library database [38](#)  
  and file locking [39](#)  
  corruption of [39](#)  
  size limit on [39](#)

Library structure [109](#)  
Library.Cell:View [109](#)  
  controlling compilation into [165](#)

License promotion [311](#)  
License queueing  
  ncsim [306](#)

License suspension [311](#)

-licqueue  
  for ncsim [306](#)

Line breakpoints [428](#)

-linedebug [150](#)

-list  
  for ncshell [389](#)  
LMC Hardware Modeling Interface [948](#)  
LMSI  
  see Logic Modeling Corporation  
    Hardware Modeling  
    Interface [948](#)  
-load [565](#)  
-loadcfc  
  for ncsim [306](#)  
-loadfmi  
  for ncsim [307](#)  
-loadpli1 [203](#)  
-loadvhpi  
  for ncsim [307](#)  
-loadvpi [205](#)  
  for ncsim [308](#)  
LOG\_FILE [792](#)  
-logfile  
  for ncelab [206](#)  
  for ncexport [836](#)  
  for ncls [855](#)  
  for ncpack [863](#)  
  for ncrm [903](#)  
  for ncsdfc [909](#)  
  for ncshell [917](#)  
  for ncsim [308](#)  
  for ncupdate [930](#)  
  for ncvlog [150](#)

Logic Modeling Corporation Hardware  
  Modeling Interface [948](#)

## M

Macros  
  defining [178](#)  
  undefining [696](#)  
Manuals  
  related to NC-Verilog [45](#)  
MAP\_INNER [821](#)  
Mapping  
  escaped names to file system  
    names [113](#)  
master.tag file [124, 138, 160](#)  
-maxdelays [207](#)  
Memory  
  dumping VHDL [563](#)  
  loading VHDL [563](#)  
memory command [563](#)  
  examples [566](#)

modifiers [565](#)  
     -dump [566](#)  
     -load [565](#)  
 options [565](#)  
     -end [566](#)  
     -file [565](#)  
     -start [566](#)  
 syntax [565](#)  
**Memory requirements** [30](#)  
**Memory usage**  
     displaying [623](#)  
**Menu commands**  
     Control  
         Run [325](#)  
     File  
         Commands  
             Source [335](#)  
         Exit [338](#)  
         Find  
             Text [489](#)  
     Set  
         Probe [422](#)  
**Messages**  
     printing extended VHDL assert [303](#)  
     suppressing VHDL assert [486](#)  
     timing check violation [743](#)  
**-messages** [400](#)  
     for ncelab [207](#)  
     for ncexport [836](#)  
     for ncls [855](#)  
     for ncpack [863](#)  
     for ncrm [903](#)  
     for ncsdfc [909](#)  
     for ncshell [917](#)  
     for ncsim [309](#)  
     for ncupdate [930](#)  
     for ncvlog [150](#)  
**-mindelays** [208](#)  
**Mixed-language**  
     and SDF annotation [830](#)  
**Mixed-language designs**  
     compiling  
         a Verilog-XL design [393](#)  
         compiling a Leapfrog VMI model [399](#)  
**Model Manager**  
     for OMI models [952](#)  
**Model manager**  
     for Quickturn [967](#)  
**Modifiers**  
     alias command  
         -set [498](#)  
     -unset [498](#)  
**check command**  
     -delete [510](#)  
     -disable [510](#)  
     -enable [510](#)  
     -show [511](#)  
**database command**  
     -close [525](#)  
     -disable [524](#)  
     -enable [525](#)  
     -open [520](#)  
     -setdefault [524](#)  
     -show [524](#)  
**fmibkpt command** [550](#)  
     -disable [550](#)  
     -enable [550](#)  
     -show [550](#)  
**memory command** [565](#)  
     -dump [566](#)  
     -load [565](#)  
**omi command**  
     -list [569](#)  
     -send [569](#)  
**probe command**  
     -create [574](#)  
     -delete [580](#)  
     -disable [580](#)  
     -enable [581](#)  
     -save [581](#)  
     -show [581](#)  
**restart command**  
     -show [596](#)  
**scope command**  
     -describe [609](#)  
     -drivers [609](#)  
     -list [610](#)  
     -set [610](#)  
     -show [610](#)  
**stack command**  
     -set [618](#)  
     -show [619](#)  
**stop command**  
     -create [626](#)  
     -delete [631](#)  
     -disable [632](#)  
     -enable [632](#)  
     -show [632](#)  
**task command**  
     -schedule [652](#)  
**MODULE** [820](#)  
**-module**

for ncls [856](#)  
Module path delays  
    and distributed delays [784](#)  
    and strength changes [785](#)  
    assigning delays [771](#)  
    calculating values for x transitions [774](#)  
    describing paths [759](#)  
    driving wired logic outputs [785](#)  
    edge-sensitive paths [761](#)  
    full connection [768](#)  
    multiple delays for a path [775](#)  
    overview [755](#)  
    parallel connection [768](#)  
    PATHPULSE\$ [283](#)  
    pulse filtering style [285](#)  
    setting pulse control for specific  
        paths [283](#)  
    setting pulse controls [280](#)  
    simple paths [760](#)  
    simulating path outputs that drive other  
        path outputs [786](#)  
    specify properties [776](#)  
        pathdelay\_controlsignal [776](#)  
        pathdelay\_max0 [776](#)  
        pathdelay\_max1 [776](#)  
        pathdelay\_sense [776](#)  
    state-dependent paths [763](#)  
Monitoring value changes [578](#)  
MTM [817](#)  
MTM\_CONTROL [792](#)  
Multi-source interconnect delays  
    (VITAL) [795](#)

**N**

Native compiled code [25](#)  
Navigator [671](#)  
-nbasync  
    for ncsm [309](#)  
NC Verilog  
    memory requirements [30](#)  
    overview of [25](#)  
ncelab [181](#)  
    hdl.var variables [232](#)  
ncelab command  
    examples [227](#)  
    options [188](#)  
        -access [188](#)  
        -afile [189](#)  
        -anno\_simtime [189](#)

-append\_log [190](#)  
-binding [190](#)  
-cdslib [190](#)  
-compile [191](#)  
-conffile [191](#)  
-conflat [194](#)  
-confname [194](#)  
-coverage [194](#)  
-delay\_mode [195](#)  
-disable\_enht [196](#)  
-epulse\_neg [196](#)  
-epulse\_ondetect [196](#)  
-epulse\_onevent [197](#)  
-errormax [197](#)  
-extend\_tcheck\_data\_limit [197](#)  
-  
    extend\_tcheck\_reference\_lim  
        it [198](#)  
-file [199](#)  
-genafile [200](#)  
-generic [200](#)  
-hdlvar [201](#)  
-help [202](#)  
-ieee1364 [202](#)  
-intermod\_path [202](#)  
-loadpli1 [203](#)  
-loadvpi [205](#)  
-logfile [206](#)  
-maxdelays [207](#)  
-messages [207](#)  
-mindelays [208](#)  
-ncerror [208](#)  
-ncfatal [209](#)  
-neg\_tchk [209](#)  
-neverwarn [209](#)  
-no\_tchk\_msg [213](#)  
-no\_tchk\_xgen [213](#)  
-no\_vpd\_msg [213](#)  
-no\_vpd\_xgen [213](#)  
-noautosdf [210](#)  
-nocopyright [210](#)  
-nodeadcode [210](#)  
-noipd [210](#)  
-nolog [211](#)  
-noneg\_tchk [211](#)  
-nonotifier [211](#)  
-nostdout [212](#)  
-notimingchecks [212](#)  
-novitalaccl [212](#)  
-nowarn [212](#)  
-ntc\_warn [214](#)

## NC-Verilog Simulator Help

---

-overwrite [214](#)  
-pathpulse [215](#)  
-plinowarn [215](#)  
-prompt [216](#)  
-pulse\_e [217](#)  
-pulse\_int\_e [217](#)  
-pulse\_int\_r [217](#)  
-pulse\_r [217](#)  
-relax [217](#)  
-sdf\_cmd\_file [219](#)  
-sdf\_no\_warnings [219](#)  
-sdf\_nocheck\_celltype [219](#)  
-sdf\_precision [220](#)  
-sdf\_verbose [220](#)  
-sdf\_worstcase\_rounding [220](#)  
-snapshot [221](#)  
-status [221](#)  
-tfile [222](#)  
-timescale [222](#)  
-typdelays [223](#)  
-update [223](#)  
-use5x4vhdl [223](#)  
-usearch [224](#)  
-v93 [224](#)  
-version [225](#)  
-vipdmax [224](#)  
-vipdmin [225](#)  
-work [226](#)  
syntax [185](#)  
-ncelabfile [401](#)  
**NCELABOPTS** variable [122](#)  
-ncerror  
    for ncelab [208](#)  
    for ncexport [836](#)  
    for nchelp [844](#)  
    for ncls [856](#)  
    for ncpack [863](#)  
    for ncrm [903](#)  
    for ncsdfc [909](#)  
    for ncshell [917](#)  
    for ncsim [310](#)  
    for ncsuffix [926](#)  
    for ncupdate [931](#)  
    for ncvlog [151](#)  
    for ncximport [401](#)  
ncexport [833](#)  
ncexport command  
    examples [838](#)  
    options [835](#)  
        -append\_log [835](#)  
        -cdslib [835](#)  
-context [835](#)  
-directory [835](#)  
-errormax [835](#)  
-exclude [835](#)  
-hdlvar [836](#)  
-help [836](#)  
-include [836](#)  
-logfile [836](#)  
-messages [836](#)  
-ncerror [836](#)  
-ncfatal [837](#)  
-nocopyright [837](#)  
-nolog [837](#)  
-nostdout [837](#)  
-overwrite [837](#)  
-snapshot [837](#)  
-target [837](#)  
-version [838](#)  
syntax [834](#)  
-ncfatal  
    for ncelab [209](#)  
    for ncexport [837](#)  
    for nchelp [844](#)  
    for ncls [856](#)  
    for ncpack [863](#)  
    for ncrm [903](#)  
    for ncsdfc [909](#)  
    for ncshell [917](#)  
    for ncsim [310](#)  
    for ncsuffix [926](#)  
    for ncupdate [931](#)  
    for ncvlog [151](#)  
    for ncximport [401](#)  
nchelp [44, 842](#)  
nchelp command  
    examples [845](#)  
    options [843](#)  
        -all [843](#)  
        -cdslib [843](#)  
        -hdlvar [843](#)  
        -help [844](#)  
        -ncerror [844](#)  
        -ncfatal [844](#)  
        -neverwarn [844](#)  
        -nocopyright [844](#)  
        -nowarn [845](#)  
        -tools [845](#)  
        -version [845](#)  
    syntax [45, 843](#)  
NCHELP\_DIR variable [122](#)  
NCLaunch [48](#)

## NC-Verilog Simulator Help

---

ncls [847](#)  
ncls command  
    examples [860](#)  
    options [853](#)  
        -all [853](#)  
        -append\_log [853](#)  
        -architecture [853](#)  
        -body [853](#)  
        -cdllib [853](#)  
        -code [853](#)  
        -command [853](#)  
        -configuration [854](#)  
        -dependents [854](#)  
        -entity [854](#)  
        -file [854](#)  
        -hdlvar [854](#)  
        -help [855](#)  
        -library [855](#)  
        -logfile [855](#)  
        -messages [855](#)  
        -module [856](#)  
        -ncerror [856](#)  
        -ncfatal [856](#)  
        -neverwarn [857](#)  
        -nocopyright [857](#)  
        -nolog [857](#)  
        -nostdout [857](#)  
        -nowarn [857](#)  
        -overlay [857](#)  
        -package [857](#)  
        -primitive [858](#)  
        -snapshot [858](#)  
        -source [858](#)  
        -time [858](#)  
        -verilog [858](#)  
        -version [859](#)  
        -vhdl [859](#)  
        -view [859](#)  
    syntax [851](#)  
ncpack [861](#)  
ncpack command  
    examples [866](#)  
    options [862](#)  
        -addonly [862](#)  
        -append\_log [862](#)  
        -cdllib [862](#)  
        -database [862](#)  
        -errormax [862](#)  
        -hdlvar [863](#)  
        -help [863](#)  
        -logfile [863](#)  
    syntax [851](#)  
ncprep [867](#)  
    and PLI [883](#)  
    and SDF annotation [883](#)  
    and Verilog-XL Dash (-) Options [878](#)  
    and Verilog-XL Plus (+) Options [879](#)  
    example output [874](#)  
    example run [872](#)  
    troubleshooting [887](#)  
ncprep command  
    options [869](#)  
        +debug [869](#)  
        +linedebug [869](#)  
        +ncerror+ [869](#)  
        +ncfatal+ [870](#)  
        +nclibdirname+ [870](#)  
        +overwrite [871](#)  
        +redirect+ [871](#)  
        -h [869](#)  
        -l [869](#)  
    syntax [868](#)  
ncrm [901](#)  
ncrm command  
    examples [904](#)  
    options [902](#)  
        -append\_log [902](#)  
        -cdllib [902](#)  
        -force [902](#)  
        -hdlvar [902](#)  
        -help [902](#)  
        -library [902](#)  
        -logfile [903](#)  
        -messages [903](#)  
        -ncerror [903](#)  
        -ncfatal [903](#)  
        -neverwarn [903](#)  
        -nocopyright [904](#)  
        -nolog [904](#)

## NC-Verilog Simulator Help

---

-nostdout [904](#)  
-nowarn [904](#)  
-snapshot [904](#)  
-version [904](#)  
syntax [901](#)  
**ncsdfc [906](#)  
**ncsdfc command**  
  examples [911](#)  
  options [908](#)  
    -append\_log [908](#)  
    -cdslib [908](#)  
    -compile [908](#)  
    -cputime [908](#)  
    -decompile [908](#)  
    -hdlvar [908](#)  
    -help [908](#)  
    -logfile [909](#)  
    -messages [909](#)  
    -ncerror [909](#)  
    -ncfatal [909](#)  
    -neverwarn [909](#)  
    -nocopyright [910](#)  
    -nolog [910](#)  
    -nostdout [910](#)  
    -output [910](#)  
    -status [910](#)  
    -stdin [910](#)  
    -update [910](#)  
    -version [911](#)  
    -worstcase\_rounding [911](#)  
syntax [907](#)  
**NCSDFCOPTS variable [123](#)  
**ncshell**  
  for FMI, SWIFT, or LMSI import [912](#)  
  for Verilog or VHDL import [385](#)  
  syntax  
    for Verilog and VHDL import [385](#)  
**ncshell command**  
  options  
    -all [914](#)  
    -analopts [387](#)  
    -analyze [387](#)  
    -append\_log [915](#)  
    -backward [915](#)  
    -cdslib [388](#)  
    -comp [388, 915](#)  
    -errormax [915](#)  
    -file [915](#)  
    -fmient [916](#)  
    -fmilib [916](#)  
    -generic [388](#)  
-hdlvar [389](#)  
-help [916](#)  
-import [916](#)  
-import verilog [389](#)  
-import vhdl [389](#)  
-into [916](#)  
-into verilog [389](#)  
-into vhdl [389](#)  
-list [389](#)  
-logfile [917](#)  
-messages [917](#)  
-ncerror [917](#)  
-ncfatal [917](#)  
-neverwarn [917](#)  
-nocompile [918](#)  
-nocopyright [918](#)  
-noescape [391](#)  
-nolog [918](#)  
-nostdout [918](#)  
-nowarn [918](#)  
-shell [391, 918](#)  
-suffix [392, 918](#)  
-ulogic [392](#)  
-version [919](#)  
-view [392, 919](#)  
-work [392](#)  
syntax  
  for FMI, SWIFT, LMSI [913](#)  
**ncsim [297](#)  
  hdl.var variables [321](#)  
  invoking [322, 335](#)  
**ncsim command**  
  examples [320](#)  
  options [301](#)  
    -append\_log [301](#)  
    -batch [301](#)  
    -cdslib [302](#)  
    -epulse\_no\_msg [302](#)  
    -errormax [302](#)  
    -exit [303](#)  
    -extassertmsg [303](#)  
    -file [303](#)  
    -gui [304](#)  
    -hdlvar [304](#)  
    -help [305](#)  
    -input [305](#)  
    -keyfile [305](#)  
    -licqueue [306](#)  
    -loadfcf [306](#)  
    -loadfmi [307](#)  
    -loadvhpi [307](#)******

-loadvpi [308](#)  
-logfile [308](#)  
-messages [309](#)  
-nbasync [309](#)  
-ncerror [310](#)  
-ncfatal [310](#)  
-neverwarn [310](#)  
-nocopyright [311](#)  
-nokey [311](#)  
-nolcpromote [311](#)  
-nolcsuspend [311](#)  
-nolog [312](#)  
-nosource [312](#)  
-nostdout [313](#)  
-nowarn [313](#)  
-plinoptwarn [314](#)  
-plinowarn [314](#)  
-ppe [314](#)  
-profile [315](#)  
-profthread [315](#)  
-redmem [315](#)  
-run [315](#)  
-status [316](#)  
-tcl [316](#)  
-unbuffered [317](#)  
-update [317](#)  
-vcdexend [318](#)  
-version [318](#)  
-xlstyle\_units [318](#)  
syntax [299](#)  
-ncsimfile [401](#)  
NCSIMOPTS variable [123](#)  
NCSIMRC variable [123](#)  
ncsuffix [925](#)  
ncsuffix command  
    examples [927](#)  
    options [925](#)  
        -ast [925](#)  
        -cod [926](#)  
        -help [926](#)  
        -ncerror [926](#)  
        -ncfatal [926](#)  
        -nocopyright [926](#)  
        -pak [926](#)  
        -sig [926](#)  
        -sss [927](#)  
        -version [927](#)  
        -vst [927](#)  
    syntax [925](#)  
ncupdate [928](#)  
ncupdate command  
examples [933](#)  
options [929](#)  
    -append\_log [929](#)  
    -cdslib [929](#)  
    -errormax [929](#)  
    -exclfile [929](#)  
    -exclude [930](#)  
    -force [930](#)  
    -hdlvar [930](#)  
    -help [930](#)  
    -ieee [930](#)  
    -library [930](#)  
    -logfile [930](#)  
    -messages [930](#)  
    -ncerror [931](#)  
    -ncfatal [931](#)  
    -neverwarn [931](#)  
    -nocopyright [931](#)  
    -nolog [931](#)  
    -norecompile [932](#)  
    -nosource [932](#)  
    -nostdout [932](#)  
    -nowarn [932](#)  
    -overwrite [932](#)  
    -script [932](#)  
    -show [932](#)  
    -unit [932](#)  
    -verbose [933](#)  
    -version [933](#)  
syntax [928](#)  
NCUPDATEOPTS variable [123](#)  
NCUSE5X variable [124](#)  
ncverilog  
    access to simulation objects [51](#)  
    and PLI [85](#)  
    and SDF annotation [85](#)  
    command options [56](#)  
    command syntax [55](#)  
    example run [75](#)  
    how it works [53](#)  
    loading a saved snapshot [65](#)  
ncverilog command  
options  
    +all [56](#)  
    +cdslib+ [56](#)  
    +checkargs [56](#)  
    +compile [56](#)  
    +debug [56](#)  
    +elaborate [57](#)  
    +expand [57](#)  
    +hdlvar+ [57](#)

```

+import 57
+mixedlang 57
+name+ 58
+ncelabargs+ 58
+ncelabexe+ 59
+ncerror+ 59
+ncfatal+ 60
+ncldirname+ 60
+ncls_all 61
+ncls_dependents 61
+ncls_snapshots 62
+ncls_source 62
+ncsimargs+ 62
+ncsimexe+ 63
+ncuid+ 63
+ncversion 63
+ncvlogargs+ 63
+noautosdf 64
+noupdate 64
+ppe 64
+sdf_orig_dir 65
+work+ 65
-help 57
-R 65
-r 65
plus options 66
NCVERILOGOPTS variable 124
NCVHDLOPTS variable 124
ncvlog 138
    hdl.var variables 160
ncvlog command
    examples 158
    options 141
        -append_log 141
        -cdslib 141
        -checktasks 143
        -define 143
        -errormax 143
        -file 144
        -hdlvar 146
        -help 146
        -ieee1364 146
        -inmdir 147
        -lexpragma 149
        -libcell 149
        -linedebug 150
        -logfile 150
        -messages 150
        -ncerror 151
        -ncfatal 151
        -neverwarn 151
    -nocopyright 152
    -noline 152
    -nolog 152
    -nomempack 152
    -nopragmawarn 153
    -nostdout 153
    -nowarn 153
    -pragma 153
    -specificunit 154
    -status 154
    -unit 154
    -uppercase 155
    -update 155
    -use5x 155
    -version 156
    -view 156
    -work 156
syntax 140
NCVLOGOPTS variable 125
ncximport command
    options 400
        -append_log 400
        -errormax 400
        -help 400
        -logfile 400
        -messages 400
        -ncelabfile 401
        -ncerror 401
        -ncfatal 401
        -ncsimfile 401
        -neverwarn 401
        -nocopyright 402
        -nolog 402
        -nostdout 402
        -novxlopts 402
        -nowarn 402
        -version 402
        -work 402
    syntax 400
ncximport utility 399
-neg_tchk 209
NETDELAY keyword 1018
-neverwarn 401
    for ncelab 209
    for ncelpack 844
    for ncls 857
    for ncpack 864
    for ncrm 903
    for ncsdfc 909
    for ncshell 917
    for ncsim 310

```

---

for ncupdate [931](#)  
for ncvlog [151](#)  
nmp program [113](#)  
-no\_tchk\_msg [213](#)  
-no\_tchk\_xgen [213](#)  
-no\_vpd\_msg [213](#)  
-no\_vpd\_xgen [213](#)  
-noautosdf  
    for ncelab [210](#)  
NOCHANGE keyword [1037](#)  
-nocompile  
    for ncshell [918](#)  
-nocopyright [402](#)  
    for ncelab [210](#)  
    for ncexport [837](#)  
    for nchelp [844](#)  
    for ncls [857](#)  
    for ncpack [864](#)  
    for ncrm [904](#)  
    for ncsdfc [910](#)  
    for ncshell [918](#)  
    for ncsim [311](#)  
    for ncsuffix [926](#)  
    for ncupdate [931](#)  
    for ncvlog [152](#)  
-nodeadcode  
    for ncelab [210](#)  
-noescape  
    for ncshell [391](#)  
-noipd [210](#)  
-nokey [311](#)  
-nolicpromote [311](#)  
-nolicssuspend  
    for ncsim [311](#)  
-noline [152](#)  
-nolog [402](#)  
    for ncelab [211](#)  
    for ncexport [837](#)  
    for ncls [857](#)  
    for ncpack [864](#)  
    for ncrm [904](#)  
    for ncsdfc [910](#)  
    for ncshell [918](#)  
    for ncsim [312](#)  
    for ncupdate [931](#)  
    for ncvlog [152](#)  
-nomempack  
    for ncvlog [152](#)  
-nomm\_objext  
    for shellgen [936](#)  
-noneg\_tchk [211](#)

-nonotifier [211](#)  
-nopragmawarn  
    for ncvlog [153](#)  
-norecompile  
    for ncupdate [932](#)  
noshowcancelled keyword [288](#)  
-nosource  
    for ncsim [312](#)  
    for ncupdate [932](#)  
-nostdout [402](#)  
    for ncelab [212](#)  
    for ncexport [837](#)  
    for ncls [857](#)  
    for ncpack [864](#)  
    for ncrm [904](#)  
    for ncsdfc [910](#)  
    for ncshell [918](#)  
    for ncsim [313](#)  
    for ncupdate [932](#)  
    for ncvlog [153](#)  
Notifiers [723](#)  
-notimingchecks [212](#)  
-novitalaccl [212](#)  
-novxlopts [402](#)  
-nowarn [402](#)  
    for ncelab [212](#)  
    for nchelp [845](#)  
    for ncls [857](#)  
    for ncpack [864](#)  
    for ncrm [904](#)  
    for ncshell [918](#)  
    for ncsim [313](#)  
    for ncupdate [932](#)  
    for ncvlog [153](#)  
-ntc\_warn [214](#)

## O

-object  
    for strobe [645](#)  
Object breakpoints [429](#)  
Objects  
    displaying information about [437](#)  
omi command [568](#)  
    examples [570](#)  
    modifiers [569](#)  
        -list [569](#)  
        -send [569](#)  
    options [569](#)  
        -all [569](#)

-instance [569](#)  
 -manager [569](#)  
 syntax [568](#)  
 OMI models [952](#)  
     and C++ model managers [966](#)  
     generating a shell for [954](#)  
     integrating [953](#)  
     integrating into a Verilog design [955](#)  
     integrating into a VHDL design [959](#)  
     simulating with [963](#)  
 On-Detect pulse filtering [285](#)  
 On-Event pulse filtering [285](#)  
 Online help [41](#)  
     invoking [42](#)  
 Open Model Interface [952](#)  
 Options  
     call command  
         -predefined [506](#)  
         -systf [506](#)  
     check command  
         -contention [509](#)  
         -delay [509](#)  
         -float [510](#)  
         -name [511](#)  
     database command  
         -compress [520](#)  
         -default [521](#)  
         -evcd [521](#)  
         -event [521](#)  
         -into [522](#)  
         -maxsize [522](#)  
         -shm [523](#)  
         -timescale [523](#)  
         -vcd [523](#)  
     deposit command  
         -absolute [529](#)  
         -after [529](#)  
         -inertial [529](#)  
         -relative [529](#)  
         -transport [529](#)  
     drivers command  
         -effective [537](#)  
         -future [537](#)  
         -novalue [538](#)  
         -verbose [538](#)  
     help command  
         -brief [555](#)  
         -functions [555](#)  
         -variables [555](#)  
     history command  
         keep [558](#)  
     redo [557](#)  
     substitute [558](#)  
     memory command [565](#)  
         -end [566](#)  
         -file [565](#)  
         -start [566](#)  
     ncelab command  
         -access [188](#)  
         -afile [189](#)  
         -anno\_simtime [189](#)  
         -append\_log [190](#)  
         -binding [190](#)  
         -cdlib [190](#)  
         -compile [191](#)  
         -conffile [191](#)  
         -conflat [194](#)  
         -confname [194](#)  
         -coverage [194](#)  
         -delay\_mode [195](#)  
         -disable\_enht [196](#)  
         -epulse\_neg [196](#)  
         -epulse\_ondetect [196](#)  
         -epulse\_onevent [197](#)  
         -errormax [197](#)  
         -extend\_tcheck\_data\_limit [197](#)  
         -  
             extend\_tcheck\_reference\_limi  
             t [198](#)  
         -file [199](#)  
         -genafile [200](#)  
         -generic [200](#)  
         -hdlvar [201](#)  
         -help [202](#)  
         -ieee1364 [202](#)  
         -intermod\_path [202](#)  
         -loadpli1 [203](#)  
         -loadvpi [205](#)  
         -logfile [206](#)  
         -maxdelays [207](#)  
         -messages [207](#)  
         -mindelays [208](#)  
         -ncerror [208](#)  
         -ncfatal [209](#)  
         -neg\_tchk [209](#)  
         -neverwarn [209](#)  
         -no\_tchk\_msg [213](#)  
         -no\_tchk\_xgen [213](#)  
         -no\_vpd\_msg [213](#)  
         -no\_vpd\_xgen [213](#)  
         -noautosdf [210](#)  
         -nocopyright [210](#)

## NC-Verilog Simulator Help

---

-nodeadcode [210](#)  
-noipd [210](#)  
-nolog [211](#)  
-noneg\_tchk [211](#)  
-nonotifier [211](#)  
-nostdout [212](#)  
-notimingchecks [212](#)  
-novitalaccl [212](#)  
-nowarn [212](#)  
-ntc\_warn [214](#)  
-overwrite [214](#)  
-pathpulse [215](#)  
-plinowarn [215](#)  
-prompt [216](#)  
-pulse\_e [217](#)  
-pulse\_int\_e [217](#)  
-pulse\_int\_r [217](#)  
-pulse\_r [217](#)  
-relax [217](#)  
-sdf\_cmd\_file [219](#)  
-sdf\_no\_warnings [219](#)  
-sdf\_nocheck\_celltype [219](#)  
-sdf\_precision [220](#)  
-sdf\_verbose [220](#)  
-sdf\_worstcase\_rounding [220](#)  
-snapshot [221](#)  
-status [221](#)  
-tfile [222](#)  
-timescale [222](#)  
-typdelays [223](#)  
-update [223](#)  
-use5x4vhdl [223](#)  
-usearch [224](#)  
-v93 [224](#)  
-version [225](#)  
-vipdmax [224](#)  
-vipdmin [225](#)  
-work [226](#)

ncexport command  
-append\_log [835](#)  
-cdslib [835](#)  
-context [835](#)  
-directory [835](#)  
-errormax [835](#)  
-exclude [835](#)  
-hdlvar [836](#)  
-help [836](#)  
-include [836](#)  
-logfile [836](#)  
-messages [836](#)  
-ncerror [836](#)

-ncfatal [837](#)  
-nocopyright [837](#)  
-nolog [837](#)  
-nostdout [837](#)  
-overwrite [837](#)  
-snapshot [837](#)  
-target [837](#)  
-version [838](#)

nchelp command  
-all [843](#)  
-cdslib [843](#)  
-hdlvar [843](#)  
-help [844](#)  
-ncerror [844](#)  
-ncfatal [844](#)  
-neverwarn [844](#)  
-nocopyright [844](#)  
-nowarn [845](#)  
-tools [845](#)  
-version [845](#)

ncls command  
-all [853](#)  
-append\_log [853](#)  
-architecture [853](#)  
-body [853](#)  
-cdslib [853](#)  
-code [853](#)  
-command [853](#)  
-configuration [854](#)  
-dependents [854](#)  
-entity [854](#)  
-file [854](#)  
-hdlvar [854](#)  
-help [855](#)  
-library [855](#)  
-logfile [855](#)  
-messages [855](#)  
-module [856](#)  
-ncerror [856](#)  
-ncfatal [856](#)  
-neverwarn [857](#)  
-nocopyright [857](#)  
-nolog [857](#)  
-nostdout [857](#)  
-nowarn [857](#)  
-overlay [857](#)  
-package [857](#)  
-primitive [858](#)  
-snapshot [858](#)  
-source [858](#)  
-time [858](#)

## NC-Verilog Simulator Help

---

-verilog [858](#)  
-version [859](#)  
-vhdl [859](#)  
-view [859](#)

ncpack command  
-addonly [862](#)  
-append\_log [862](#)  
-cdslib [862](#)  
-database [862](#)  
-errormax [862](#)  
-hdlvar [863](#)  
-help [863](#)  
-logfile [863](#)  
-messages [863](#)  
-ncerror [863](#)  
-ncfatal [863](#)  
-neverwarn [864](#)  
-nocopyright [864](#)  
-nolog [864](#)  
-nostdout [864](#)  
-nowarn [864](#)  
-readonly [864](#)  
-tmpdir [864](#)  
-unlock [865](#)  
-unpack [865](#)  
-version [865](#)

ncprep command  
+debug [869](#)  
+linedebug [869](#)  
+ncerror+ [869](#)  
+ncfatal+ [870](#)  
+nclibdirname+ [870](#)  
+overwrite [871](#)  
+redirect+ [871](#)  
-h [869](#)  
-l [869](#)

ncrm command  
-append\_log [902](#)  
-cdslib [902](#)  
-force [902](#)  
-hdlvar [902](#)  
-help [902](#)  
-library [902](#)  
-logfile [903](#)  
-messages [903](#)  
-ncerror [903](#)  
-ncfatal [903](#)  
-neverwarn [903](#)  
-nocopyright [904](#)  
-nolog [904](#)  
-nostdout [904](#)

-nowarn [904](#)  
-snapshot [904](#)  
-version [904](#)

ncsdfc  
-update [910](#)

ncsdfc command  
-append\_log [908](#)  
-cdslib [908](#)  
-compile [908](#)  
-cputime [908](#)  
-decompile [908](#)  
-hdlvar [908](#)  
-help [908](#)  
-logfile [909](#)  
-messages [909](#)  
-ncerror [909](#)  
-ncfatal [909](#)  
-neverwarn [909](#)  
-nocopyright [910](#)  
-nolog [910](#)  
-nostdout [910](#)  
-output [910](#)  
-status [910](#)  
-stdin [910](#)  
-version [911](#)  
-worstcase\_rounding [911](#)

ncshell command  
-all [914](#)  
-analopts [387](#)  
-analyze [387](#)  
-append\_log [915](#)  
-backward [915](#)  
-cdslib [388](#)  
-comp [388, 915](#)  
-errormax [915](#)  
-file [915](#)  
-fmient [916](#)  
-fmilib [916](#)  
-generic [388](#)  
-hdlvar [389](#)  
-help [916](#)  
-import [916](#)  
-import verilog [389](#)  
-import vhdl [389](#)  
-into [916](#)  
-into verilog [389](#)  
-into vhdl [389](#)  
-list [389](#)  
-logfile [917](#)  
-messages [917](#)  
-ncerror [917](#)

## NC-Verilog Simulator Help

---

-ncfatal [917](#)  
-neverwarn [917](#)  
-nocompile [918](#)  
-nocopyright [918](#)  
-noescape [391](#)  
-nolog [918](#)  
-nostdout [918](#)  
-nowarn [918](#)  
-shell [391, 918](#)  
-suffix [392, 918](#)  
-ulogic [392](#)  
-version [919](#)  
-view [392, 919](#)  
-work [392](#)

ncsim command  
-append\_log [301](#)  
-batch [301](#)  
-cdslib [302](#)  
-epulse\_no\_msg [302](#)  
-errormax [302](#)  
-exit [303](#)  
-extassertmsg [303](#)  
-file [303](#)  
-gui [304](#)  
-hdlvar [304](#)  
-help [305](#)  
-input [305](#)  
-keyfile [305](#)  
-licqueue [306](#)  
-loadcfc [306](#)  
-loadfmi [307](#)  
-loadvhpi [307](#)  
-loadvpi [308](#)  
-logfile [308](#)  
-messages [309](#)  
-nbasync [309](#)  
-ncerror [310](#)  
-ncfatal [310](#)  
-neverwarn [310](#)  
-nocopyright [311](#)  
-nokey [311](#)  
-nolcpromote [311](#)  
-nolicsuspend [311](#)  
-nolog [312](#)  
-nosource [312](#)  
-nostdout [313](#)  
-nowarn [313](#)  
-plinooptwarn [314](#)  
-plinowarn [314](#)  
-ppe [314](#)  
-profile [315](#)

-profthread [315](#)  
-redmem [315](#)  
-run [315](#)  
-status [316](#)  
-tcl [316](#)  
-unbuffered [317](#)  
-update [317](#)  
-vcdextend [318](#)  
-version [318](#)  
-xlstyle\_units [318](#)

ncsuffix command  
-ast [925](#)  
-cod [926](#)  
-help [926](#)  
-ncerror [926](#)  
-ncfatal [926](#)  
-nocopyright [926](#)  
-pak [926](#)  
-sig [926](#)  
-sss [927](#)  
-version [927](#)  
-vst [927](#)

ncupdate command  
-append\_log [929](#)  
-cdslib [929](#)  
-errormax [929](#)  
-exclfile [929](#)  
-exclude [930](#)  
-force [930](#)  
-hdlvar [930](#)  
-help [930](#)  
-ieee [930](#)  
-library [930](#)  
-logfile [930](#)  
-messages [930](#)  
-ncerror [931](#)  
-ncfatal [931](#)  
-neverwarn [931](#)  
-nocopyright [931](#)  
-nolog [931](#)  
-norecompile [932](#)  
-nosource [932](#)  
-nostdout [932](#)  
-nowarn [932](#)  
-overwrite [932](#)  
-script [932](#)  
-show [932](#)  
-unit [932](#)  
-verbose [933](#)  
-version [933](#)

ncverilog command

```

+all 56
+cdslib+ 56
+checkargs 56
+compile 56
+debug 56
+elaborate 57
+hdlvar+ 57
+import 57
+mixedlang 57
+name+ 58
+ncelabargs+ 58
+ncelabexe+ 59
+ncerror+ 59
+ncfatal+ 60
+ncldirname+ 60
+ncls_all 61
+ncls_dependents 61
+ncls_snapshots 62
+ncls_source 62
+ncsimargs+ 62
+ncsimexe+ 63
+ncuid+ 63
+ncversion 63
+ncvlogargs+ 63
+noautosdf 64
+noupdate 64
+ppe 64
+sdf_orig_dir 65
+work+ 65
-help 57
-R 65
-r 65
ncvlog command
-append_log 141
-cdslib 141
-checktasks 143
-define 143
-errormax 143
-file 144
-hdlvar 146
-help 146
-ieee1364 146
-incdir 147
-lexpragma 149
-libcell 149
-linedebug 150
-logfile 150
-messages 150
-ncerror 151
-ncfatal 151
-neverwarn 151
nocopyright 152
-noline 152
-nolog 152
-nomempack 152
-nopragmawarn 153
-nostdout 153
-nowarn 153
-pragma 153
-specificunit 154
-status 154
-unit 154
-upcase 155
-update 155
-use5x 155
-version 156
-view 156
-work 156
ncximport command
-append_log 400
-errormax 400
-help 400
-logfile 400
-messages 400
-ncelabfile 401
-ncerror 401
-ncfatal 401
-ncsimfile 401
-neverwarn 401
-nocopyright 402
-nolog 402
-nostdout 402
-novxlopts 402
-nowarn 402
-version 402
-work 402
omi command
-all 569
-instance 569
-manager 569
probe command
-all 575
-database 575, 581
-depth 576
-evcd 576
-inputs 577
-name 577
-outputs 577
-ports 578
-screen 578
-shm 579
-variables 579

```

-vcd [579](#)  
 -waveform [580](#)  
 process command  
   -all [588](#)  
   -current [588](#)  
   -eot [588](#)  
   -next [588](#)  
 release command  
   -keepvalue [593](#)  
 run command  
   -clean [599](#)  
   -delta [599](#)  
   -next [599](#)  
   -phase [599](#)  
   -process [599](#)  
   -return [600](#)  
   -step [600](#)  
   -timepoint [600](#)  
 save command  
   -commands [604](#)  
   -environment [604](#)  
   -overwrite [604](#)  
   -simulation [604](#)  
 scope command  
   -names [609](#)  
   -sort [609](#)  
   -up [610](#)  
 shellgen command [935](#)  
   +quickturn\_mm\_option [939](#)  
   -b [935](#)  
   -f [936](#)  
   -help [936](#)  
   including model manager invocation  
     options [938](#)  
   -l [936](#)  
   -m [936](#)  
   -nomm\_objext [936](#)  
   -nomm\_path [937](#)  
   -o [937](#)  
   -pli [937](#)  
   -r [937](#)  
   -unresolved [938](#)  
   -verilog [938](#)  
   -version [938](#)  
   -vhdl [938](#)  
 stack command  
   -down [619](#)  
   -up [619](#)  
 stop command  
   -condition [627](#)  
   -continue [627](#)  
   -delbreak [627](#)  
   -delta [627](#)  
   -execute [628](#)  
   -if [628](#)  
   -line [629](#)  
   -name [629](#)  
   -object [629](#)  
   -process [630](#)  
   -silent [630](#)  
   -skip [630](#)  
   -subprogram [631](#)  
   -time [631](#)  
 strobe command  
   -condition [645](#)  
   -delete [646](#)  
   -help [646](#)  
   -object [645](#)  
   -time [645](#)  
 time command  
   -delta [656](#)  
   -nounit [656](#)  
 -output  
   for ncsdfc [910](#)  
 -overlay  
   for nccls [857](#)  
 Overview  
   of NC Verilog [25](#)  
 -overwrite [604](#)  
   for ncelab [214](#)  
   for ncexport [837](#)  
   for ncupdate [932](#)  
 OVI 2.0  
   compliance with [90](#)

## P

pack\_assert\_off variable [483](#)  
 -package  
   for nccls [857](#)  
 Packages  
   suppressing assert messages from [486](#)  
 -pak  
   for ncsuffix [926](#)  
 Parallel connection  
   in module path delay [768](#)  
 Path  
   delays [1009](#)  
 PATH keyword  
   in access file [257](#)  
   in timing access file [268](#)

**P**ATHCONSTRAINT keyword [1039](#)  
 pathdelay\_controlsignal [776](#)  
 pathdelay\_max0 [776](#)  
 pathdelay\_max1 [776](#)  
 pathdelay\_sense [776](#)  
 -pathpulse [215](#)  
**PATHPULSE** keyword [1023](#)  
**PATHPULSE\$** [283](#)  
**PATHPULSEPERCENT** keyword [1024](#)  
**Paths**  
 describing [759](#)  
**pc.db** file [124](#), [138](#), [160](#)  
**Performance**  
 and coding style [674](#)  
**PERIOD** keyword [1036](#)  
**PERIODCONSTRAINT** keyword [1041](#)  
**PLI** [941](#)  
**PLI tasks**  
 checking for [143](#)  
**PLI1.0**  
 loading applications with -loadpli1 [203](#)  
**-plinoptwarn** [314](#)  
**-plinowarn**  
 for ncelab [215](#)  
 for ncsim [314](#)  
**PORT** keyword [1014](#)  
**Post Processing Environment**  
 for SimVision [672](#)  
 invoking [314](#)  
**-ppe**  
 for ncsim [314](#)  
**-pragma**  
 for ncvlog [153](#)  
**Pragmas**  
 enabling processing of [153](#)  
 lexical [149](#)  
 turning off warning messages [153](#)  
**Precision**  
 specifying [220](#)  
**Predefined Tcl variables** [482](#)  
 assert\_1164\_warnings [482](#)  
 assert\_report\_level [482](#)  
 assert\_stop\_level [482](#)  
 autoscope [482](#)  
 clean [483](#)  
 display\_unit [483](#)  
 pack\_assert\_off [483](#)  
 severity\_pack\_assert\_off [484](#)  
 snapshot [484](#)  
 tcl\_prompt1 [484](#)  
 tcl\_prompt2 [485](#)  
 time\_scale [485](#)  
 time\_unit [485](#)  
 vhdl\_format [485](#)  
 vlog\_format [485](#)  
**Preferences form** [489](#)  
**-primitive**  
 for ncls [858](#)  
**Printing online documents** [43](#)  
**probe command** [571](#)  
 examples [582](#)  
 modifiers [574](#)  
     -[create](#) [574](#)  
     -[delete](#) [580](#)  
     -[disable](#) [580](#)  
     -[enable](#) [581](#)  
     -[save](#) [581](#)  
     -[show](#) [581](#)  
 options [574](#)  
     -[all](#) [575](#)  
     -[database](#) [575](#), [581](#)  
     -[depth](#) [576](#)  
     -[evcd](#) [576](#)  
     -[inputs](#) [577](#)  
     -[name](#) [577](#)  
     -[outputs](#) [577](#)  
     -[ports](#) [578](#)  
     -[screen](#) [578](#)  
     -[shm](#) [579](#)  
     -[variables](#) [579](#)  
     -[vcf](#) [579](#)  
     -[waveform](#) [580](#)  
 syntax [573](#)  
**Process breakpoints** [431](#)  
**process command** [587](#)  
 examples [589](#)  
 options [588](#)  
     -[all](#) [588](#)  
     -[current](#) [588](#)  
     -[eot](#) [588](#)  
     -[next](#) [588](#)  
**process command syntax** [587](#)  
**PROCESS** keyword [1002](#)  
**Profile**  
 generating [315](#)  
**-profile**  
 for ncsim [315](#)  
**-profthread**  
 for ncsim [315](#)  
**PROGRAM** keyword [1002](#)  
**Programming Language Interface**  
 (PLI) [941](#)

-prompt  
    for ncelab [216](#)  
Pulse controls [280](#)  
Pulse filtering  
    and cancelled schedules [288](#)  
Pulse filtering style [285](#)  
-pulse\_e [217](#)  
-pulse\_int\_e [217](#)  
-pulse\_int\_r [217](#)  
-pulse\_r [217](#)  
pulsestyle\_ondetect keyword [287](#)  
pulsestyle\_onevent keyword [287](#)

## Q

Queueing  
    ncsim license [306](#)  
Quickturn [967](#)  
    and \$omiCommand system task [973](#)  
    clocked mode [972](#)  
    clocked mode options [972](#)  
    creating gate-level netlist [969](#)  
    event mode [971](#)  
    generating emulator database [970](#)  
    generating simulation shell file [970](#)  
    model manager for [967](#)  
    plus options for shell generator [970](#)  
    restrictions on [975](#)

## R

-R  
    for ncverilog [65](#)  
-r  
    for ncverilog [65](#)  
Radix  
    setting a default [480](#)  
readmemb  
    See \$readmemb [1061](#)  
readmemh  
    See \$readmemh [1061](#)  
-readonly  
    for ncpack [864](#)  
recordabort  
    See \$recordabort [446](#)  
recordclose  
    See \$recordclose [446](#)  
recordfile  
    See \$recordfile [446](#)

recordoff  
    See \$recordoff [446](#)  
recordon  
    See \$recordon [446](#)  
recordsetup  
    See \$recordsetup [446](#)  
recordvars  
    See \$recordvars [446](#)  
RECOVERY keyword [1030](#)  
RECREM keyword [1033](#)  
-redmem [315](#)  
Regression mode [248](#)  
    and ncverilog [51](#)  
    and PLI applications [249](#)  
    and SimVision [251](#)  
    and Tcl commands [249](#)  
Reinvoking the simulation [330](#)  
Related manuals [45](#)  
-relax  
    for ncelab [217](#)  
release command [592](#)  
    examples [593](#)  
    syntax [592](#)  
release commands  
    options  
        -keepvalue [593](#)  
REMOVAL keyword [1032](#)  
Removing assigned attributes from  
    libraries [112](#)  
Replacing values in existing delays [1007](#)  
reset command [594](#)  
    examples [594](#)  
    syntax [594](#)  
Resetting the simulation [329](#)  
restart command [595](#)  
    examples [596](#)  
    modifiers  
        -show [596](#)  
        syntax [596](#)  
Restarting the simulation state [327](#)  
Restoring  
    the simulation environment [490](#)  
Restoring the simulation state [327](#)  
RETAIN keyword [1013](#)  
-run [315](#)  
run command [598](#)  
    examples [600](#)  
    options [599](#)  
        -clean [599](#)  
        -delta [599](#)  
        -next [599](#)

---

-phase [599](#)  
-process [599](#)  
-return [600](#)  
-step [600](#)  
-timepoint [600](#)  
syntax [598](#)  
using to simulate to the next line [434](#)

**S**

save command [602](#)  
examples [604](#)  
options [604](#)  
  -commands [604](#)  
  -environment [604](#)  
  -overwrite [604](#)  
  -simulation [604](#)  
syntax [604](#)

Saving  
  the simulation environment [490](#)

Saving the simulation state [327](#)

SCALE\_FACTORS [793, 817](#)  
SCALE\_TYPE [793, 818](#)  
Schematic Window [671](#)  
SCOPE [791](#)  
Scope [424](#)  
scope command [608](#)  
  examples [611](#)  
  modifiers [609](#)  
    -describe [609](#)  
    -drivers [609](#)  
    -list [610](#)  
    -set [610](#)  
    -show [610](#)  
  options [609](#)  
    -names [609](#)  
    -sort [609](#)  
    -up [610](#)  
syntax [608](#)  
traversing hierarchy with [424](#)

-script  
  for ncupdate [932](#)

Script files  
  executing with input commnd [560](#)

SDF  
  and ncverilog [85](#)  
  and Verilog [801](#)  
  annotating mixed-language [830](#)  
  annotating to VITAL [790](#)  
  annotating with [789](#)

annotator output [795](#)  
cell entries in SDF file [1003](#)  
command file [791](#)  
  COMPILED\_SDF\_FILE [791](#)  
  CONFIG\_FILE [792](#)  
  examples [793](#)  
  LOG\_FILE [792](#)  
  MTM\_CONTROL [792](#)  
  SCALE\_FACTORS [793](#)  
  SCALE\_TYPE [793](#)  
  SCOPE [791](#)  
    specifying [795](#)  
  command line options [826](#)  
  compiling an SDF file [790](#)  
  configuration file [813](#)  
    example [814](#)  
    IGNORE [815](#)  
    INTERCONNECT\_DELAY [816](#)  
    INTERCONNECT\_MIPD [817](#)  
    MAP\_INNER [821](#)  
    MODULE [820](#)  
    MTM [817](#)  
    SCALE\_FACTORS [817](#)  
    SCALE\_TYPE [818](#)  
    syntax [813](#)  
    TURNOFF\_DELAY [819](#)  
  example SDF files [1051](#)  
  keywords  
    ABSOLUTE [1007](#)  
    ARRIVAL [1045](#)  
    CELL [1003](#)  
    CELLTYPE [1004](#)  
    COND [1012, 1026](#)  
    CONDELSE [1013](#)  
    DATE [1002](#)  
    DELAY [1006](#)  
    DELAYFILE [1001](#)  
    DEPARTURE [1046](#)  
    DESIGN [1002, 1021](#)  
    DIFF [1044](#)  
    DIVIDER [1002](#)  
    GLOBALPATHPULSE [1024](#)  
    HOLD [1028](#)  
    INCLUDE [1005](#)  
    INCREMENT [1008](#)  
    INSTANCE [1004](#)  
    INTERCONNECT [1016](#)  
    IOPATH [1009](#)  
    NETDELAY [1018](#)  
    NOCHANGE [1037](#)  
    PATHCONSTRAINT [1039](#)

PATHPULSE [1023](#)  
 PATHPULSEPERCENT [1024](#)  
 PERIOD [1036](#)  
 PERIODCONSTRAINT [1041](#)  
 PORT [1014](#)  
 PROCESS [1002](#)  
 PROGRAM [1002](#)  
 RECOVERY [1030](#)  
 RECREM [1033](#)  
 REMOVAL [1032](#)  
 RETAIN [1013](#)  
 SDFVERSION [1002](#)  
 SETUP [1027](#)  
 SETUPHOLD [1029](#)  
 SKEW [1034](#)  
 SKEWCONSTRAINT [1042](#)  
 SLACK [1047](#)  
 SUM [1043](#)  
 TEMPERATURE [1002](#)  
 TIMESCALE [1003](#)  
 TIMINGCHECK [1025](#)  
 TIMINGENV [1038](#)  
 VENDOR [1002](#)  
 VERSION [1002](#)  
 VOLTAGE [1002](#)  
 WAVEFORM [1048](#)  
 WIDTH [1035](#)  
 specifying precision [220](#)  
 syntax of SDF file [994](#)  
     characters [997](#)  
     conventions [996](#)  
     header [1001](#)  
     identifiers [996](#)  
     operators [999](#)  
     overview [995](#)  
     Version 3.0 keywords [999](#)  
 turning off automatic annotation [64](#),  
[210](#)  
 -sdf\_cmd\_file [219](#)  
 -sdf\_no\_warnings [219](#)  
 -sdf\_nocheck\_celltype [219](#)  
 -sdf\_precision [220](#)  
 -sdf\_verbose [220](#)  
 -sdf\_worstcase\_rounding [220](#)  
 SDFVERSION keyword [1002](#)  
 Search order  
     for setup files  
         changing [131](#)  
 Searching  
     online documents [43](#)  
 set command [481](#)

Set menu  
     Probe [422](#)  
 Setup files  
     specifying search order for [131](#)  
 SETUP keyword [1027](#)  
 setup.loc file [131](#)  
 SETUPHOLD keyword [1029](#)  
 severity\_pack\_assert\_off variable [484](#)  
 -shell  
     for ncshell [391](#), [918](#)  
 shellgen [934](#)  
     shellgen command  
         example [939](#)  
         options [935](#)  
             +quickturn\_mm\_option [939](#)  
             -b [935](#)  
             -f [936](#)  
             -help [936](#)  
             including model manager invocation  
                 options [938](#)  
             -I [936](#)  
             -m [936](#)  
             -nomm\_objext [936](#)  
             -nomm\_path [937](#)  
             -o [937](#)  
             -pli [937](#)  
             -r [937](#)  
             -unresolved [938](#)  
             -verilog [938](#)  
             -version [938](#)  
             -vhdl [938](#)  
         syntax [935](#)  
 SHM databases  
     compressing [520](#)  
     dumping all value changes to [521](#)  
     limiting the size of [522](#)  
     using \$recordvars [446](#)  
     using system tasks to open and probe  
         signals [440](#)  
 -show  
     for ncupdate [932](#)  
     for restart command [596](#)  
 showcanceled keyword [288](#)  
 -sig  
     for ncsuffix [926](#)  
 Signal Flow Browser [671](#)  
 SimControl [671](#)  
     customizing [489](#)  
 Simple module paths [760](#)  
 Simulating  
     invoking the simulator [322](#), [335](#)

## NC-Verilog Simulator Help

---

managing databases [418](#)  
overview of [297](#)  
providing interactive commands from a file [335](#)  
starting a simulation [325](#)  
updating changes when you invoke ncsim [332](#)

Simulation objects  
    displaying information about [437](#)

Simulation state  
    saving and restarting [327](#)

Simulator commands  
    getting help on [44](#)

SimVision analysis environment  
    invoking [672](#)

SimVision debug environment [671](#)  
    post processing mode [672](#)

SimVision Waveform Viewer [440](#)  
    invoking [444](#)

Single stepping [434](#)

Size limit  
    of library database [39](#)

SKEW keyword [1034](#)

SKEWCONSTRAINT keyword [1042](#)

SLACK keyword [1047](#)

Snapshot  
    loading a saved snapshot in ncverilog [65](#)  
    simulating in ncverilog [65](#)

-snapshot  
    for ncelab [221](#)  
    for ncexport [837](#)  
    for ncls [858](#)  
    for ncrm [904](#)

snapshot [181](#)

snapshot variable [484](#)

SOFTINCLUDE statement  
    in cds.lib file [112](#)  
    in hdl.var file [121](#)

-source  
    for ncls [858](#)

source command [616](#)  
    examples [617](#)  
    syntax [616](#)

Source files  
    editing [488](#)

-specificunit [154](#)

Specify block [758](#)  
    parameters in [758](#)  
    pathdelay\_controlsignal [776](#)  
    pathdelay\_max0 [776](#)

pathdelay\_max1 [776](#)  
pathdelay\_sense [776](#)  
syntax of [758](#)

Specify properties [776](#)  
    pathdelay\_controlsignal [776](#)  
    pathdelay\_max0 [776](#)  
    pathdelay\_max1 [776](#)  
    pathdelay\_sense [776](#)

Specparam declaration [758](#)

SRC\_ROOT variable [125](#)

-sss  
    for ncsuffix [927](#)

stack command [618](#)  
    examples [619](#)  
    modifiers [618](#)  
        -set [618](#)  
        -show [619](#)  
    options [618](#)  
        -down [619](#)  
        -up [619](#)  
    syntax [618](#)

-start  
    for memory command [566](#)

State-dependent path delays [763](#)

Statistics  
    memory usage  
        displaying [623](#)

-status  
    for ncelab [221](#)  
    for ncdfc [910](#)  
    for ncsim [316](#)  
    for ncvlog [154](#)

status command [623](#)  
    examples [623](#)  
    syntax [623](#)

-stdin  
    for ncdfc [910](#)

Stepping through code [434](#)

stop command [624](#)  
    examples [633](#)  
    modifiers [626](#)  
        -create [626](#)  
        -delete [631](#)  
        -disable [632](#)  
        -enable [632](#)  
        -show [632](#)  
    options [626](#)  
        -condition [627](#)  
        -continue [627](#)  
        -delbreak [627](#)  
        -delta [627](#)

**-execute** [628](#)  
**-if** [628](#)  
**-line** [629](#)  
**-name** [629](#)  
**-object** [629](#)  
**-process** [630](#)  
**-silent** [630](#)  
**-skip** [630](#)  
**-subprogram** [631](#)  
**-time** [631](#)  
**syntax** [625](#)  
 using to disable, enable, delete, or show  
     breakpoints [433](#)  
 using to set a condition breakpoint [427](#)  
 using to set a delta breakpoint [431](#)  
 using to set a line breakpoint [428](#)  
 using to set a process breakpoint [431](#)  
 using to set a subprogram  
     breakpoint [432](#)  
 using to set a time breakpoint [430](#)  
 using to set an object breakpoint [429](#)  
**Strength changes**  
 on path inputs [785](#)  
**strobe command** [644](#)  
 examples [646](#)  
 options [645](#)  
     -condition [645](#)  
     -delete [646](#)  
     -help [646](#)  
     -object [645](#)  
     -time [645](#)  
 syntax [645](#)  
**Subprogram breakpoints** [432](#)  
**-suffix**  
     for ncshell [392, 918](#)  
**SUM keyword** [1043](#)  
**Syntax conventions for commands** [496](#)  
**System tasks**  
     \$omiCommand [965](#)  
     calling from the command line [504](#)  
     checking for non-standard [143](#)  
     for SHM databases [440](#)

**T**

**-target**  
     for ncexport [837](#)  
**task command** [651](#)  
 examples [652](#)  
 modifiers [652](#)

**-schedule** [652](#)  
**syntax** [651](#)  
**Tasks**  
 scheduling from Tcl [651](#)  
**Tcl** [976](#)  
     extensions to [982](#)  
     summary of basic syntax [976](#)  
**-tcl** [316](#)  
**Tcl commands**  
     executing a script file of [560](#)  
     getting help on [44](#)  
     using wildcards [495](#)  
**Tcl variables**  
     predefined [482](#)  
         assert\_1164\_warnings [482](#)  
         assert\_report\_level [482](#)  
         assert\_stop\_level [482](#)  
         autoscope [482](#)  
         clean [483](#)  
         display\_unit [483](#)  
         pack\_assert\_off [483](#)  
         severity\_pack\_assert\_off [484](#)  
         snapshot [484](#)  
         tcl\_prompt1 [484](#)  
         tcl\_prompt2 [485](#)  
         time\_scale [485](#)  
         time\_unit [485](#)  
         vhdl\_format [485](#)  
         vlog\_format [485](#)  
     tcl\_prompt1 variable [484](#)  
     tcl\_prompt2 variable [485](#)  
**TEMPERATURE keyword** [1002](#)  
**Text**  
     searching for [489](#)  
**-tfile**  
     for ncelab [222](#)  
**-time**  
     for ncls [858](#)  
     for strobe [645](#)  
**Time breakpoints** [430](#)  
**time command** [655](#)  
     examples [656](#)  
     options [656](#)  
         -delta [656](#)  
         -nounit [656](#)  
     syntax [655](#)  
**Time values**  
     formatting of [318](#)  
**time\_scale variable** [485](#)  
**time\_unit variable** [485](#)  
**Timescale**

setting on command line [222](#)  
**-timescale**  
     for database command [523](#)  
     for ncelab [222](#)  
**timescale**  
     specifying for VCD file [523](#)  
**TIMESCALE keyword** [1003](#)  
**Timing**  
     retaining port values [1013](#)  
**Timing access file** [222](#)  
     keywords [268](#)  
     writing [267](#)  
**Timing checks** [703](#)  
     conditioned events [726](#)  
     negative  
         turning off [211](#)  
     nonconvergence of  
         extending data limit [197](#)  
         extending reference limit [198](#)  
     overview [704](#)  
**SDF** [1025](#)  
     conditional [1026](#)  
     HOLD [1028](#)  
     NOCHANGE [1037](#)  
     PERIOD [1036](#)  
     RECOVERY [1030](#)  
     RECREM [1033](#)  
     REMOVAL [1032](#)  
     SETUP [1027](#)  
     SETUPHOLD [1029](#)  
     SKEW [1034](#)  
     WIDTH [1035](#)  
     suppressing execution of [212](#)  
     suppressing warning messages [213](#)  
**system tasks** [705](#)  
     \$hold [706](#)  
     \$nochange [722](#)  
     \$period [713](#)  
     \$recovery [716](#)  
     \$recrem [719](#)  
     \$removal [717](#)  
     \$setup [705](#)  
     \$setuphold [707](#)  
     \$skew [715](#)  
     \$width [711](#)  
     using edge-control specifiers [722](#)  
     using notifiers for timing violations [723](#)  
     violation messages [743](#)  
**TIMINGCHECK keyword** [1025](#)  
**TIMINGENV keyword** [1038](#)  
**Tk** [36](#)

**TMP attribute** [112](#)  
**TMP libraries** [114](#)  
     explicit [114](#)  
     implicit [115](#)  
**-tmpdir**  
     for ncpack [864](#)  
**Tools**  
     getting help on [43](#)  
**-tools**  
     for nchelp [845](#)  
**TURNOFF\_DELAY** [819](#)  
**-typdelays** [223](#)  
**Type conversion functions** [991](#)

## U

**-ulogic**  
     for ncshell [392](#)  
**-unbuffered** [317](#)  
**UNDEFINE statement**  
     in cds.lib file [111](#)  
     in hdl.var file [120](#)  
**Undefining libraries** [111](#)  
**Undefining variables**  
     in hdl.var file [120](#)  
**-unit**  
     for ncupdate [932](#)  
     for ncvlog [154](#)  
**UNIX commands**  
     executing [494](#)  
**-unlock**  
     for ncpack [865](#)  
**-unpack**  
     for ncpack [865](#)  
**-upcase** [155](#)  
**-update**  
     for ncelab [223](#)  
     for ncsdfc [910](#)  
     for ncsim [317](#)  
     for ncvlog [155](#)  
**Updating design changes** [332](#)  
**-use5x** [155](#)  
**-use5x4vhdl**  
     for ncelab [223](#)  
**-usearch**  
     for ncelab [224](#)  
**Utilities** [831](#)  
     ncexport [833](#)  
     nchelp [44, 842](#)  
     ncls [847](#)

ncpack [861](#)  
ncprep [867](#)  
ncrm [901](#)  
ncsdhc [906](#)  
ncshellfor FMI, SWIFT, LMSI  
    import [912](#)  
ncsuffix [925](#)  
ncupdate [928](#)  
ncxlimport [399](#)  
shellgen [934](#)

**V**

-v93  
    for ncelab [224](#)  
value command [658](#)  
    examples [660](#)  
    syntax [658](#)  
Values  
    depositing [436](#)  
Variables  
    defining in hdl.var file [120](#)  
    setting [481](#)  
    undefining in hdl.var file [120](#)  
VCD databases  
    extended VCD [463](#)  
    for mixed-language design [410](#)  
    generating [458](#)  
        using system tasks [460](#)  
        using Tcl commands [458](#)  
        limiting the size of [522](#)  
-vcextend [318](#)  
VENDOR keyword [1002](#)  
-verbose  
    for ncupdate [933](#)  
-verilog  
    for ncls [858](#)  
Verilog coding guidelines [674](#)  
Verilog Desktop simulator [28](#)  
Verilog Procedural Interface (VPI) [941](#)  
Verilog source files  
    compiling [137](#)  
verilog.v file [124, 138, 160](#)  
VERILOG\_SUFFIX variable [125](#)  
Verilog-XL [50](#)  
    compliance with [90](#)  
-version [402](#)  
    for ncelab [225](#)  
    for ncexport [838](#)  
    for nchelp [845](#)

for ncls [859](#)  
for ncpack [865](#)  
for ncrm [904](#)  
for ncsdgc [911](#)  
for ncshell [919](#)  
for ncsim [318](#)  
for ncsuffix [927](#)  
for ncupdate [933](#)  
for ncvlog [156](#)  
version command [663](#)  
    examples [663](#)  
    syntax [663](#)  
VERSION keyword [1002](#)  
-vhdl  
    for ncls [859](#)  
VHDL language rules  
    relaxing [217](#)  
vhdl\_format variable [485](#)  
VHDL\_SUFFIX variable [125](#)  
VHPI  
    loading applications with -loadvhi [307](#)  
View  
    definition of [109](#)  
-view [156](#)  
    for ncls [859](#)  
    for ncshell [392, 919](#)  
VIEW variable [126](#)  
VIEW\_MAP variable [126](#)  
Viewports [952](#)  
-vipdmax  
    for ncelab [224](#)  
-vipdmin  
    for ncelab [225](#)  
Visibility of simulation objects [248](#)  
    in ncverilog [51](#)  
VITAL [790](#)  
    multi-source interconnect delays [795](#)  
    SDF annotation [790](#)  
vlog\_format variable [485](#)  
VOLTAGE keyword [1002](#)  
VPI [941](#)  
    annotation at simulation time [189](#)  
    loading applications with -loadvpi [205, 308](#)  
-vst  
    for ncsuffix [927](#)  
VXLOPTS variable [126](#)

### W

Watch Objects Window [671](#)  
WAVEFORM keyword [1048](#)  
Waveforms  
    displaying with SimVision Waveform  
        Viewer [440](#)  
where command [664](#)  
    examples [664](#)  
    syntax [664](#)  
WIDTH keyword [1035](#)  
Wildcards  
    in Tcl commands [495](#)  
Wired logic outputs  
    and module paths [785](#)  
-work [402](#)  
    for ncclab [226](#)  
    for ncshell [392](#)  
    for ncvlog [156](#)  
WORK variable [127](#)  
-worstcase\_rounding [911](#)

### X

-xstyle\_units [318](#)