

VCS®/VCSI™ User Guide

Version C-2009.06
June 2009

Comments?
E-mail your comments about this manual to:
vcs_support@synopsys.com.

SYNOPSYS®

Copyright Notice and Proprietary Information

Copyright © 2008 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of
_____ and its employees. This is copy number _____. "

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AMPS, Cadabra, CATS, CRITIC, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSPICE, HSIM, iN-Phase, in-Sync, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PrimeTime, SiVL, SNUG, SolvNet, System Compiler, TetraMAX, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, Columbia, Columbia-CE, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, Direct Silicon Access, Discovery, Encore, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, HSIMplus, HSPICE-Link, iN-Tandem, i-Virtual Stepper, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Magellan, Mars, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, Planet, Planet-PL, Polaris, Power Compiler, Raphael, Raphael-NES, Saturn, Scirocco, Scirocco-i, Star-RCXT, Star-SimXT, Taurus, TSUPREM-4, VCS Express, VCSi, VHDL Compiler, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (SM)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.

ARM and AMBA are registered trademarks of ARM Limited.

Saber is a registered trademark of SabreMark Limited Partnership and is used under license.

All other product or company names may be trademarks of their respective owners.

Contents

1. Getting Started	
VCS Support with Technologies	1-2
Setting Up VCS	1-4
Verifying Your System Configuration	1-4
Obtaining a License	1-5
Setting Up Your Environment	1-6
Setting Up Your C Compiler	1-7
Using VCS	1-8
Basic Usage Model	1-8
2. VCS Flow	
Compilation	2-1
Using vcs	2-2
Commonly Used Options	2-2
Simulation	2-10
Interactive Mode	2-11
Batch Mode	2-11

Commonly Used Runtime Options.	2-12
3. Modeling Your Design	
Avoiding Race Conditions	3-2
Using and Setting a Value at the Same Time	3-3
Setting a Value Twice at the Same Time	3-4
Flip-Flop Race Condition	3-5
Continuous Assignment Evaluation	3-6
Counting Events.	3-7
Time Zero Race Conditions	3-8
Optimizing Testbenches for Debugging.	3-9
Conditional Compilation.	3-10
Enabling Debugging Features At Runtime.	3-11
Combining the Techniques	3-15
Avoiding the Debugging Problems From Port Coercion	3-15
Creating Models That Simulate Faster	3-16
Unaccelerated Data Types, Primitives, and Statements	3-18
Inferring Faster Simulating Sequential Devices.	3-19
Modeling Faster always Blocks	3-24
Using the +v2k Compile-Time Option	3-25
Case Statement Behavior	3-26
Precedence in Text Macro Definitions.	3-27
Memory Size Limits in VCS.	3-27
Using Sparse Memory Models	3-28

Obtaining Scope Information	3-29
Scope Format Specifications	3-29
Returning Information About the Scope	3-33
Avoiding Circular Dependency	3-37
Designing With \$lsi_dumpports for Simulation and Test	3-38
Dealing With Unassigned Nets	3-39
Code Values at Time 0	3-40
Cross Module Forces and No Instance Instantiation	3-41
Signal Value/Strength Codes	3-43
4. Compiling the Design	
Compiling the Design in Debug Mode	4-1
Compiling the Design in Optimized Mode	4-3
Key Compilation Features	4-3
Initializing Verilog Memories and Registers	4-4
Usage Model	4-5
Overriding Parameters	4-5
Checking for X and Z Values In Conditional Expressions	4-7
Enabling the Checking	4-8
Filtering Out False Negatives	4-9
VCS V2K Configurations and Libmaps	4-11
Library Mapping Files	4-12
Configurations	4-13
Hierarchical Configurations	4-16
The -top Compile-time Option	4-17
Limitations of Configurations	4-18

Using +evalorder Option	4-18
5. Simulating the Design	
Using DVE.....	5-2
Using UCLI	5-3
Key Runtime Features.....	5-4
Passing Values from the Runtime Command Line	5-5
Profiling the Simulation	5-6
CPU Time Views	5-7
Memory Usage Views	5-16
Save and Restart The Simulation	5-19
Save and Restart Example.....	5-20
Save and Restart File I/O.....	5-21
Save and Restart With Runtime Options	5-22
Specifying a Long Time Before Stopping The Simulation	5-24
How VCS Prevents Time 0 Race Conditions.....	5-26
6. VCS Multicore Technology	
Assertion Level Processing	
VCS Multicore Technology Options.....	6-30
Use Model for Assertion Simulation.....	6-32
Use Model for Toggle and Functional Coverage	6-32
Use Model for VPD Dumping.....	6-32
Running VCS Multicore Simulation	6-32
Assertion Simulation	6-33
Toggle Coverage	6-33

Functional Coverage	6-35
VPD File.....	6-37
7. VPD, VCD, and EVCD Utilities	
Advantages of VPD	7-2
Dumping a VPD File	7-3
Using System Tasks.....	7-3
Enable and Disable Dumping.....	7-4
Override the VPD Filename	7-8
Dump Multi-dimensional Arrays and Memories	7-9
Using \$vcplusmemorydump	7-20
Capture Delta Cycle Information	7-20
Dumping an EVCD File	7-21
Post-processing Utilities	7-23
The vcdpost Utility	7-24
Scalarizing the Vector Signals	7-25
Uniquifying the Identifier Codes	7-26
The vcdpost Utility Syntax	7-26
The vcdiff Utility	7-27
The vcdiff Utility Syntax	7-28
The vcat Utility	7-36
The vcat Utility Syntax	7-36
Generating Source Files From VCD Files	7-40
Writing the Configuration File	7-42
The vcsplit Utility	7-46
The vcsplit Utility Syntax	7-47
The vcd2vpd Utility	7-50

Options for specifying EVCD options	7-52
The vpd2vcf Utility	7-53
The Command File Syntax	7-60
The vpdmerge Utility	7-64
8. Using Discovery Visual Environment	
General Requirements	8-2
DVE Options	8-2
Help Options	8-2
Post-processing, Interactive, and Script Options	8-3
64-bit Platform Support	8-4
Working with the DVE	8-4
Managing Target Views	8-6
Populating Other Views and Panes	8-7
Displaying Variables in the Data Pane	8-7
Dragging and Dropping Scopes	8-7
Using the Menu Bar and Toolbar	8-9
Running a Simulation	8-10
Setting Up and Starting an Interactive Session	8-10
Navigating the Design Hierarchy	8-11
Working with Scopes and Signals	8-13
Viewing Values	8-13
Viewing Signal Source Code	8-13
Dumping Signal Values	8-14
Forcing Signal Values	8-15
Working with Signal Groups	8-15

Creating Signal Groups	8-16
Managing Signal Groups	8-17
Displaying Signal Groups	8-18
Merging Signal Groups	8-18
Working with the Source Code	8-19
Managing Breakpoints in Interactive Simulation	8-20
Annotating Values	8-23
Using Waveforms	8-24
Loading Waveform Views	8-25
Using the Signal Pane	8-25
Using the Waveform Pane	8-30
Shifting Signals	8-35
Printing Waveforms	8-36
Using Schematics	8-37
Opening a Schematic View	8-38
Selecting a Signal in the Schematic	8-40
Annotating Values	8-41
Displaying Connections in a Path Schematic	8-41
Manually Tracing Signals	8-43
Searching for Signals	8-44
Following a Signal Across Boundaries	8-45
Tracing X Values in a Design	8-46
Printing Schematics	8-48
Working with Assertions and Cover Properties	8-50
Viewing Assertion Results	8-50
Setting Display Criteria	8-51
Debugging Assertions	8-53
Viewing an Assertion in the Wave view	8-54
Viewing Source Code	8-59

9. Using Unified Command-line Interface

Invoking UCLI	9-2
UCLI Related Compile-time and Runtime Options	9-2
Commonly Used UCLI Commands	9-4
Using the help Command	9-4
Tool Environment Array Commands	9-5
Controlling the Simulation	9-5
Dumping a VPD File	9-7
Setting Breakpoints	9-8
Forcing and Releasing Signals	9-10
Calling System Task or Function	9-13
Macro Control Routine.....	9-13
Navigation Command	9-14
Design Query Commands	9-15

10. Performance Tuning

Compile-time Performance	10-2
Incremental Compilation	10-3
Compile Once and Run Many Times.....	10-4
Parallel Compilation.....	10-4
Runtime Performance	10-5
Using Radiant Technology	10-5
Compiling With Radiant Technology.....	10-5
Applying Radiant Technology to Parts of the Design	10-6
Improving Performance When Using PLIs.....	10-16
Usage Model	10-17

Impact on Performance	10-18
11. Gate-level Simulation	
SDF Annotation	11-2
Using \$sdf_annotate System Task	11-2
Delays and Timing	11-4
Transport and Inertial Delays	11-5
Different Inertial Delay Implementations	11-6
Enabling Transport Delays	11-9
Pulse Control	11-10
Pulse Control with Transport Delays	11-12
Pulse Control with Inertial Delays	11-14
Specifying Pulse on Event or Detect Behavior	11-18
Specifying the Delay Mode	11-23
Using the Configuration File to Disable Timing	11-25
Using the timopt Timing Optimizer	11-25
Editing the timopt.cfg File	11-28
Editing Potential Sequential Device Entries	11-28
Editing Clock Signal Entries	11-29
Negative Timing Checks	11-30
The Need for Negative Value Timing Checks	11-31
The \$setuphold Timing Check Extended Syntax	11-36
Negative Timing Checks for Asynchronous Controls	11-40
The \$recrrem Timing Check Syntax	11-40
Enabling Negative Timing Checks	11-43
Other Timing Checks Using the Delayed Signals	11-45

Checking Conditions	11-49
Toggling the Notifier Register.....	11-50
SDF Back-annotation to Negative Timing Checks.....	11-51
How VCS Calculates Delays	11-52
Using Multiple Non-overlapping Violation Windows.....	11-54
12. Coverage	
Code Coverage	12-2
Line Coverage	12-3
Condition Coverage.....	12-4
Toggle Coverage	12-6
Finite State Machine (FSM) Coverage.....	12-6
Branch Coverage (for Verilog Only Designs).....	12-8
Usage Model	12-10
Compilation and Runtime Options.....	12-10
Unified Report Generator (URG).....	12-13
Usage Model to Invoke URG.....	12-13
Basic URG Options	12-14
Examples	12-15
13. Using OpenVera Native Testbench	
Usage Model	13-3
Example	13-3
Usage Model	13-5
Using Template Generator	13-5
Example	13-6

Key Features	13-18
Multiple Program Support	13-19
Configuration File Model	13-19
Configuration File	13-19
Usage Model for Multiple Programs	13-21
NTB Options and the Configuration File	13-22
Separate Compilation of Testbench Files	13-23
Usage Model	13-25
Example	13-25
Class Dependency Source File Reordering	13-26
Circular Dependencies	13-28
Dependency-based Ordering in Encrypted Files	13-28
Using Encrypted Files	13-29
Functional Coverage	13-30
Unified Coverage Directory and Database Control	13-30
Loading Coverage Data	13-32
Using Reference Verification Methodology	13-35
Limitations	13-36
Performance Profiler	13-36
Enabling the NTB Profiler	13-37
Performance Profiler Example	13-38
Memory Profiler	13-43
Usage Model	13-44
UCLI Interface	13-45
VCS Dynamic Memory Profile Report	13-45
14. Using SystemVerilog	
Usage Model	14-2

Using VMM with SV.....	14-2
Debugging SystemVerilog Designs.....	14-3
Functional Coverage.....	14-4
Unified Coverage Directory and Database Control	14-4
Loading Coverage Data.....	14-6
Using -cov_disable_cg to Disable Functional Coverage Items	14-9
Memory Profiler.....	14-11
Syntax	14-11
Usage Model	14-12
Memory Profile Report.....	14-13
Extensions to SystemVerilog.....	14-18
Unique/Priority Case/IF Final Semantic Enhancements	14-18
Using Unique/Priority Case/If with Always Block or Continous Assign	14-19
Using Unique/Priority Inside a Function.....	14-23
System Tasks to Control Warning Messages.....	14-25
15. Aspect Oriented Extensions	
Aspect-Oriented Extensions in SV.....	15-3
Processing of AOE as a Precompilation Expansion	15-6
Weaving advice into the target method	15-11
Pre-compilation Expansion details.....	15-16
Precedence	15-17
16. Constraints Debugging and Profiling	
Using Constraint Profiling	16-2

Using the Constraint Profiling Report	16-4
Using the Hierarchical Constraint Debugger Report	16-6
Color Coding Constraint Blocks and rand vars	16-7
Avoiding Duplicate Printing of Original Constraint Set	16-7
Array and XMR Support in std::randomize()	16-8
Error Conditions	16-10
XMR Support in Constraints	16-11
 17. OpenVera-SystemVerilog Testbench Interoperability	
Scope of Interoperability	17-2
Importing OpenVera types into SystemVerilog	17-3
Data Type Mapping	17-6
Mailboxes and Semaphores	17-7
Events	17-10
Strings	17-10
Enumerated Types	17-10
Integers and Bit-Vectors	17-13
Arrays	17-13
Structs and Unions	17-15
Connecting to the Design	17-16
Mapping Modports to Virtual Ports	17-16
Virtual Modports	17-16
Importing Clocking Block Members into a Modport	17-17
Semantic Issues with Samples, Drives, and Expects	17-23
Notes to Remember	17-23

Blocking Functions in OpenVera	17-23
Constraints and Randomization	17-24
Functional Coverage	17-24
Usage Model	17-25
Limitations	17-26
18. Integrating VCS with Debussy	
Setting Up Debussy.....	18-1
Usage Model to Dump fsdb File	18-2
Using Verilog System Tasks.....	18-2
Using UCLI.....	18-2
19. Using SystemVerilog Assertions	
Using SVAs in the HDL Design	19-2
Using Standard Checker Library	19-2
Instantiating SVA Checkers in Verilog	19-3
Binding SVA to a Design.....	19-3
Inlining SVAs in the Verilog Design	19-4
Usage Model	19-6
Controlling SystemVerilog Assertions	19-6
Compilation and Runtime Options	19-7
Assertion Monitoring System Tasks.....	19-9
Using Assertion Categories	19-13
Using OpenVera Assertion System Tasks	19-13
Using Attributes	19-14
Stopping and Restarting Assertions By Category	19-16

Viewing Results	19-21
Using a Report File	19-21
20. Using Property Specification Language	
Including PSL in the Design	20-2
Examples	20-2
Usage Model	20-3
Examples	20-3
Using SVA Options, SVA System Tasks, and OV Classes	20-4
Limitations	20-5
21. Using SystemC	
Overview	21-4
Verilog Design Containing Verilog Modules and SystemC	
Leaf Modules	21-5
Usage Model	21-6
Input Files Required	21-7
Generating Verilog Wrappers for SystemC Modules	21-8
Supported Port Data Types	21-11
Example	21-12
Controlling Time Scale and Resolution in a SystemC	21-14
Automatic adjustment of the time resolution	21-15
Setting time scale/resolution of Verilog/VHDL kernel	21-15
Setting time scale/resolution of SystemC kernel	21-16
Adding a Main Routine for Verilog-On-Top Designs	21-17
SystemC Designs Containing Verilog Modules	21-18

Usage Model	21-18
Input Files Required.	21-19
Generating a SystemC Wrapper for Verilog Modules	21-20
Example.	21-22
Compilation Scheme	21-24
SystemC Only Designs	21-27
Usage Model	21-27
Restrictions	21-28
Supported and Unsupported UCLI/DVE and CBug Features	21-29
Controlling TimeScale Resolution	21-30
Setting Timescale of SystemC Kernel	21-30
Automatic Adjustment of Time Resolution	21-31
Considerations for Export DPI Tasks.	21-32
Use syscan -export_DPI [function-name]	21-32
Use syscan -export_DPI [Verilog-file]	21-33
Use a Stubs File	21-35
Using options -Mlib and -Mdir	21-35
Specifying Runtime Options to the SystemC Simulation.	21-36
Using a Port Mapping File	21-37
Using a Data Type Mapping File	21-38
Combining SystemC with Verilog Configurations	21-40
Verilog-on-top, SystemC and/or VHDL down.	21-41
Compiling a Verilog/SystemC design	21-42
Compiling a Verilog/SystemC+VHDL design	21-43
SystemC-on-top, Verilog and/or VHDL down.	21-43

Compiling a SystemC/Verilog design	21-46
Compiling a SystemC/Verilog+VHDL design	21-47
Limitations	21-47
 Parameters	21-48
Parameters in Verilog	21-49
Parameters in VHDL	21-49
Parameters in SystemC	21-50
Verilog-on-Top, SystemC-down	21-50
VHDL-on-Top, SystemC-down	21-51
SystemC-on-Top, Verilog/VHDL down	21-52
Namespace	21-53
Parameter specification as vcs elaboration arguments	21-53
Debug	21-54
Limitations	21-54
 Debugging Mixed Simulations Using DVE or UCLI	21-55
 Transaction Level Interface	21-56
Interface Definition File	21-58
Generation of the TLI Adapters	21-61
Transaction Debug Output	21-62
Instantiation and Binding	21-63
Supported Data Types of Formal Arguments	21-66
Miscellaneous	21-67
 Delta-cycles	21-68
 Using a Customized SystemC Installation	21-69
Compatibility with OSCI SystemC	21-71

Selecting SystemC 2.0.1, 2.1 or 2.2	21-71
Compiling Source Files	21-72
Using Posix threads or quickthreads.....	21-72
Extensions.....	21-74
Installing VG GNU Package	21-78
Static and Dynamic Linking	21-78
Static Linking in VCS	21-78
Dynamic Linking in VCS.....	21-79
Dynamic Linking in VCS.....	21-80
LD_LIBRARY_PATH Environment Variable	21-81
Limitations	21-81
Verilog wrapper needed for pure VHDL-top-SystemC down	21-82

22. C Language Interface

Using PLI	22-2
Writing a PLI Application	22-2
Functions in a PLI Application	22-4
Header Files for PLI Applications.....	22-5
PLI Table File	22-6
Syntax	22-6
Using the PLI Table File	22-19
Enabling ACC Capabilities.....	22-20
Globally	22-20
Using the Configuration File.....	22-21
Selected ACC Capabilities	22-24

Using VPI Routines	22-28
Support for the vpi_register_systf Routine.	22-29
Integrating a VPI Application With VCS.	22-30
PLI Table File for VPI Routines	22-31
Unimplemented VPI Routines	22-32
Using DirectC	22-34
Using Direct C/C++ Function Calls	22-35
How C/C++ Functions Work in a Verilog Environment.	22-38
Declaring the C/C++ Function	22-40
Calling the C/C++ Function	22-46
Storing Vector Values in Machine Memory.	22-48
Converting Strings	22-50
Avoiding a Naming Problem.	22-54
Using Direct Access.	22-54
Using the vc_hdrs.h File.	22-62
Access Routines for Multi-Dimensional Arrays	22-63
Using Abstract Access.	22-64
Using vc_handle.	22-64
Using Access Routines	22-66
Summary of Access Routines	22-115
Enabling C/C++ Functions.	22-120
Mixing Direct And Abstract Access	22-122
Specifying the DirectC.h File	22-123
Extended BNF for External Function Declarations	22-123
 23. SAIF Support	
Using SAIF Files	23-2

SAIF System Tasks	23-2
Flow to Dump the Backward SAIF File	23-5
Criteria for Choosing Signals for SAIF Dumping.....	23-7
24. Encrypting Source Files	
Using Compiler Directives or Pragmas	24-2
Example.....	24-3
Using Automatic Protection Options	24-5
25. Integrating VCS with Vera	
Setting Up Vera and VCS	25-2
Using Vera with VCS.....	25-2
Usage Model	25-3
26. Integrating VCS with HSIM	
Environment Setup	26-2
Usage Model	26-2
27. Integrating VCS with NanoSim	
Environment Setup	27-2
Usage Model	27-3
28. Integrating VCS with Specman	
Type Support.....	28-2

Usage Flow	28-2
Setting Up The Environment	28-2
Specman e Code Accessing Verilog	28-3
Using specrun and specview.....	28-4
Adding Specman Objects To DVE	28-6
29. Integrating VCS with Denali	
Setting Up Denali Environment for VCS	29-1
Integrating Denali with VCS	29-2
Usage Model	29-2
Usage Model for Verilog Memory Models	29-2
Execute Denali Commands at UCLI Prompt	29-3
30. Integrating VCS with XA	
Environment Setup	30-2
Usage Model	30-2
Appendix A. VCS Environment Variables	
Simulation Environment Variables.....	A-1
Optional Environment Variables	A-3
Appendix B. Compile-time Options	
Options for Accessing Verilog Libraries.....	B-4
Options for Incremental Compilation	B-7
Options for Help and Documentation.....	B-8

Options for SystemVerilog Assertions	B-8
Options for Native Testbench	B-12
Options for Different Versions of Verilog	B-16
Options for Initializing Memories and Regs	B-17
Options for Using Radiant Technology	B-17
Options for 64-bit Compilation	B-18
Options for Starting Simulation Right After Compilation	B-18
Options for Compiling For Coverage Metrics	B-18
Options for Debugging Using DVE and UCLI	B-26
Options for Specifying Delays and SDF File	B-28
Options for Specify Blocks and Timing Checks	B-31
Options for Pulse Filtering	B-32
Options for Negative Timing Checks	B-33
Options for Profiling Your Design	B-34
Options to Specify Verilog Source Files and Compile-time Options in a File	B-34
Options for Compiling Runtime Options into the Executable	B-35
Options for PLI Applications	B-36
Options to Enable the VCS DirectC Interface	B-38
Options for Flushing Certain Output Text File Buffers	B-38
Options for Simulating SWIFT VMC Models and SmartModels	B-39
Options for Controlling Messages	B-40
Options for Cell Definition	B-43
Options for Licensing	B-44
Options for Controlling the Linker	B-45
Options for Controlling the C Compiler	B-45

Options for Source Protection	B-48
Options for Mixed Analog/Digital Simulation	B-50
Options for Changing Parameter Values	B-51
Checking for X and Z Values in Conditional Expressions	B-52
Options to Specify the Time Scale	B-52
General Options	B-53
Enable Verilog 2001 Features	B-53
Enable the VCS/SystemC Cosimulation Interface	B-53
TetraMAX	B-54
Make Accessing an Undeclared Bit an Error Condition	B-54
Allow Inout Port Connection Width Mismatches	B-54
Allow Zero or Negative Multiconcat Multiplier	B-55
Specifying a VCD File	B-55
Memories and Multi-Dimensional Arrays (MDAs)	B-55
Specifying a Log File	B-56
Changing Source File Identifiers to Upper Case	B-56
Defining a Text Macro	B-57
Specifying the Name of the Executable File	B-57
Returning The Platform Directory Name	B-58
For Long Calls	B-58
Enable Loop Detect	B-58

Appendix C. Simulation Options

Options for Simulating Native Testbenches	C-2
Options for SystemVerilog Assertions	C-5
Options for Coverage Metrics	C-11
Options for Enabling and Disabling Specify Blocks	C-14
Options for Specifying When Simulation Stops	C-15

Options for Recording Output	C-15
Options for Controlling Messages	C-15
Options for Discovery Visual Environment and UCLI	C-17
Options for VPD Files	C-17
Options for VCD Files	C-19
Options for Specifying Delays	C-20
Options for Flushing Certain Output Text File Buffers	C-22
Options for Licensing	C-23
General Options	C-24
Viewing the Compile-Time Options	C-24
Recording Where ACC Capabilities are Used	C-24
Suppressing the \$stop System Task	C-25
Enabling User-defined Plusarg Options	C-25
Enabling Overriding the Timing of a SWIFT SmartModel..	C-25
Specifying acc_handle_simulated_net PLI Routine	C-26

Appendix D. Compiler Directives and System Tasks

Compiler Directives	D-1
Compiler Directives for Cell Definition	D-2
Compiler Directives for Setting Defaults	D-2
Compiler Directives for Macros	D-3
Compiler Directives for Delays.	D-5
Compiler Directives for Backannotating SDF Delay Values..	D-6
Compiler Directives for Source Protection.	D-7
Compiler Directives for Controlling Port Coercion	D-7
General Compiler Directives	D-8
Compiler Directive for Including a Source File	D-8

Compiler Directive for Setting the Time Scale	D-8
Compiler Directive for Specifying a Library	D-9
Compiler Directive for File Names and Line Numbers....	D-10
Unimplemented Compiler Directives	D-10
 System Tasks and Functions.....	D-10
System Tasks for SystemVerilog Assertions Severity	D-10
System Tasks for SystemVerilog Assertions Control.....	D-11
System Tasks for SystemVerilog Assertions	D-12
System Tasks for VCD Files	D-12
System Tasks for LSI Certification VCD and EVCD Files ..	D-15
System Tasks for VPD Files.....	D-18
System Tasks for SystemVerilog Assertions	D-26
System Tasks for Executing Operating System Commands .	D-27
System Tasks for Log Files	D-28
System Tasks for Data Type Conversions.....	D-28
System Tasks for Displaying Information.....	D-29
System Tasks for File I/O.....	D-30
System Tasks for Loading Memories.....	D-32
System Tasks for Time Scale.....	D-33
System Tasks for Simulation Control.....	D-33
System Tasks for Timing Checks.....	D-34
System Tasks for PLA Modeling	D-37
System Tasks for Stochastic Analysis	D-37
System Tasks for Simulation Time.....	D-38
System Tasks for Probabilistic Distribution	D-39
System Tasks for Resetting VCS.....	D-39

General System Tasks and Functions	D-40
Checks for a Plusarg	D-40
SDF Files	D-40
Counting the Drivers on a Net	D-41
Depositing Values	D-41
Fast Processing Stimulus Patterns	D-41
Saving and Restarting The Simulation State	D-41
Checking for X and Z Values in Conditional Expressions	D-42
IEEE Standard System Tasks Not Yet Implemented	D-43

Appendix E. PLI Access Routines

Access Routines for Reading and Writing to Memories	E-2
acc_setmem_int	E-5
acc_getmem_int	E-6
acc_clearmem_int	E-7
Examples	E-7
acc_setmem_hexstr	E-13
Examples	E-14
acc_getmem_hexstr	E-18
acc_setmem_bitstr	E-19
acc_getmem_bitstr	E-20
acc_handle_mem_by_fullname	E-21
acc_readmem	E-22
Examples	E-23
acc_getmem_range	E-24
acc_getmem_size	E-25
acc_getmem_word_int	E-26

acc_getmem_word_range	E-27
Access Routines for Multidimensional Arrays	E-28
tf_mdanodeinfo and tf_imdanodeinfo.	E-29
acc_get_mda_range	E-31
acc_get_mda_word_range()	E-32
acc_getmda_bitstr()	E-34
acc_setmda_bitstr()	E-35
Access Routines for Probabilistic Distribution	E-36
vcs_random	E-37
vcs_random_const_seed.	E-38
vcs_random_seed	E-38
vcs_dist_uniform	E-39
vcs_dist_normal.	E-40
vcs_dist_exponential	E-41
vcs_dist_poisson	E-42
Access Routines for Returning a Pointer to a Parameter Value	E-42
acc_fetch_paramval_str.	E-43
Access Routines for Extended VCD Files	E-43
acc_lsi_dumports_all	E-45
acc_lsi_dumports_call	E-46
acc_lsi_dumports_close.	E-48
acc_lsi_dumports_flush	E-49
acc_lsi_dumports_limit.	E-50
acc_lsi_dumports_misc	E-52
acc_lsi_dumports_off.	E-53

acc_lsi_dumports_on	E-55
acc_lsi_dumports_setformat	E-56
acc_lsi_dumports_vhdl_enable	E-58
Access Routines for Line Callbacks	E-59
acc_mod_lcb_add	E-60
acc_mod_lcb_del	E-63
acc_mod_lcb_enabled	E-65
acc_mod_lcb_fetch	E-66
acc_mod_lcb_fetch2	E-67
acc_mod_sfi_fetch	E-69
Access Routines for Source Protection	E-71
vcsSpClose	E-76
vcsSpEncodeOff	E-76
vcsSpEncodeOn	E-78
vcsSpEncoding	E-79
vcsSpGetFilePtr	E-80
vcsSpInitialize	E-81
vcsSpOvaDecodeLine	E-82
vcsSpOvaDisable	E-83
vcsSpOvaEnable	E-85
vcsSpSetDisplayMsgFlag	E-86
vcsSpSetFilePtr	E-87
vcsSpSetLibLicenseCode	E-88
vcsSpSetPliProtectionFlag	E-89
vcsSpWriteChar	E-90
vcsSpWriteString	E-91

Access Routine for Signal in a Generate Block.....	E-93
acc_object_of_type	E-93
VCS API Routines.....	E-94
Vcsinit()	E-94
VcsSimUntil()	E-94

1

Getting Started

VCS® is a high-performance, high-capacity Verilog® simulator that incorporates advanced, high-level abstraction verification technologies into a single open native platform.

VCS is a compiled code simulator. It enables you to analyze, compile, and simulate Verilog, SystemVerilog, OpenVera and SystemC design descriptions. It also provides you with a set of simulation and debugging features to validate your design. These features provide capabilities for source-level debugging and simulation result viewing.

VCS accelerates complete system verification by delivering the fastest and highest capacity Verilog simulation for RTL functional verification.

This chapter describes the following sections:

- “[VCS Support with Technologies](#)”
- “[Setting Up VCS](#)”
- “[Using VCS](#)”

VCS Support with Technologies

VCS supports the following IEEE standards:

- The Verilog language as defined in the *Standard Verilog Hardware Description Language* (IEEE Std 1364).
- The IEEE Std 1800 language (with some exceptions) as defined in *SystemVerilog Language Reference Manual* for VCS/VCS MX.

In addition to its standard Verilog and SystemVerilog compilation and simulation capabilities, VCS includes the following integrated set of features and tools:

- SystemC - VCS / SystemC Co-simulation Interface enables VCS and the SystemC modeling environment to work together when simulating a system described in the Verilog and SystemC languages. For more information, refer to “[Using SystemC](#)” on page 22-1.
- Discovery Visualization Environment (DVE) — the new graphical debugging environment. You can use DVE to trace signals of interest while viewing annotated values in the source code or schematic diagrams. You can also compare waveforms, extract specific signal information, and generate testbenches based on waveform outputs. For more information, refer to “[Using Discovery Visual Environment](#)” on page 8-1.

- Unified Command-line Interface (UCLI) — Unified Command-line Interface (UCLI) provides a common set of commands for interactive simulation. UCLI is compatible with Tcl 8.3 and, therefore, any Tcl command can be used at the UCLI prompt. For more information, refer to “[Using Unified Command-line Interface](#)” on page 9-1.
- Built-In Coverage Metrics — a comprehensive built-in coverage analysis functionality that includes condition, toggle, line, finite-state-machine (FSM), path, and branch coverage. You can use coverage metrics to determine the quality of coverage of your verification test and focus on creating additional test cases. You only need to compile once to run both simulation and coverage analysis. For more information, refer to “[Coverage](#)” on page 12-1.
- DirectC Interface — this interface allows you to directly embed user-created C/C++ functions within your Verilog design description. This results in a significant improvement in ease-of-use and performance over existing PLI-based methods. VCS atomically recognizes C/C++ function calls and integrates them for simulation, thus eliminating the need to manually create PLI files.

VCS supports Synopsys DesignWare IPs, VCS Verification Library, VMC models, Vera, HSIM, and NanoSim. For information on integrating VCS with HSIM, refer to the HSIM-VCS DKI and HSIM-VCS-MX DKI Mixed-Signal Simulation Application Note. For information on integrating VCS with NanoSim, refer to the "Discovery AMS: Mixed-Signal Simulation User Guide" available in the NanoSim installation directory.

VCS can also be integrated with third-party tools such as Specman, Debussy, Denali, and other acceleration and emulation systems.

Setting Up VCS

This section outlines the basic steps for preparing to run VCS. It includes the following topics:

- “[Verifying Your System Configuration](#)”
 - “[Obtaining a License](#)”
 - “[Setting Up Your Environment](#)”
 - “[Setting Up Your C Compiler](#)”
-

Verifying Your System Configuration

You can use the `syschk.sh` script to check if your system and environment match the QSC requirements for a given release of a Synopsys product. The QSC (Qualified System Configurations) represents all system configurations maintained internally and tested by Synopsys.

To check whether the system you are on meets the QSC requirements, enter:

```
% syschk.sh
```

When you encounter any issue, run the script with tracing enabled to capture the output and contact Synopsys. To enable tracing, you can either uncomment the `set -x` line in the `syschk.sh` file or enter the following command:

```
% sh -x syschk.sh >& syschk.log
```

Use `syschk.sh -v` to generate a more verbose output stream including the exact path for various binaries used by the script, etc. For example:

```
% syschk.sh -v
```

Note:

If you copy the `syschk.sh` script to another location before using it, you must also copy the `syschk.dat` data file to the same directory.

You can also refer to the "Supported Platforms and Products" section of the VCS Release Notes for a list of supported platforms, and recommended C compiler and linker versions.

Obtaining a License

You must have a license to run VCS. To obtain a license, contact your local Synopsys Sales Representative. Your Sales Representative will need the hostid for your machine.

To start a new license, do the following:

1. Verify that your license file is functioning correctly:

```
% lmcksum -c license_file_pathname
```

Running this licensing utility ensures that the license file is not corrupt. You should see an "OK" for every INCREMENT statement in the license file.

Note:

The snpslmd platform binaries and accompanying FlexLM utilities are shipped separately and are not included with this distribution. You can download these binaries as part of the Synopsys Common Licensing (SCL) kit from the Synopsys Web Site at:

<http://www.synopsys.com/cgi-bin/ASP/sk/smartkeys.cgi>

2. Start the license server:

```
% lmgrd -c license_file_pathname -l logfile_pathname
```

3. Set the LM_LICENSE_FILE environment variable to point to the license file. For example:

```
% setenv LM_LICENSE_FILE /u/edatools/vcs/license.dat
```

Setting Up Your Environment

To run VCS, you need to set the following environment variables:

- \$VCS_HOME environment variable

Set the environment variable VCS_HOME to the path where VCS is installed as shown below:

```
% setenv VCS_HOME installation_path
```

- \$PATH environment variable

Set your UNIX PATH variable to \$VCS_HOME/bin as shown below:

```
% set path = ($VCS_HOME/bin $path)
```

OR

```
% setenv PATH $VCS_HOME/bin:$PATH
```

- LM_LICENSE_FILE environment variable

Set the license variable LM_LICENSE_FILE to your license file as shown below:

```
% setenv LM_LICENSE_FILE Location_to_the_license_file
```

For additional information on environment variables, see [Appendix A, "VCS Environment Variables"](#).

Setting Up Your C Compiler

On Solaris and Linux, VCS requires a C compiler to compile the intermediate files, and to link the executable file that you simulate. Solaris does not include a C compiler, therefore, you must purchase the C compiler for Solaris or use gcc. For Solaris, VCS assumes the C compiler is located in its default location (/usr/ccs/bin).

Linux and IBM RS/6000 AIX platforms all include a C compiler, and VCS assumes the compiler is located in its default location (/usr/bin).

You can specify a different C compiler using the environment VCS_CC or the -cc compile-time option.

Using VCS

VCS uses the following steps to compile and simulate Verilog designs:

- Compiling the Design
- Simulating the Design

Compiling the Design

VCS provides you with the `vcs` executable to compile and elaborate the design. This executable compiles your design using the intermediate files in the design or work library, generates the object code, and statically links them to generate a binary simulation executable, `simv`. For more information, see [Chapter 2, "VCS Flow"](#).

Simulating the Design

Simulate your design by executing the binary simulation executable, `simv`. For more information, see [Chapter 2, "VCS Flow"](#).

Basic Usage Model

Compilation

```
% vcs [compile_options] Verilog_files
```

Simulation

```
% simv [run_options]
```

2

VCS Flow

Simulating a design using VCS involves two basic steps:

- “[Compilation](#)”
- “[Simulation](#)”

VCS uses the same two steps to compile any design irrespective of the HDL, HVL, and other supported technologies used. For information on supported technologies, refer to “[VCS Support with Technologies](#)” on page 1-2.

Compilation

Compiling is the first step to simulate your design. In this phase, VCS builds the instance hierarchy and generates a binary executable `simv`. This binary executable is later used for simulation.

In this phase, you can choose to compile the design either in optimized mode or in debug mode. Runtime performance of VCS is based on the mode you choose and the level of flexibility required during simulation. Synopsys recommends you use full-debug or partial-debug mode until the design correctness is achieved, and then switch to optimized mode.

In optimized mode, also called batch mode, VCS delivers the best compile-time and runtime performance for a design. You typically choose optimized mode to run regressions, or when you do not require extensive debug capabilities. For more information, refer to the section “[Compiling the Design in Optimized Mode](#)”.

You compile the design in debug mode, also called interactive mode, when you are in the initial phase of your development cycle, or when you need more debug capabilities or tools to debug the design issues. In this mode, the performance will not be the best that VCS can deliver. However, using some of the compile-time options, you can compile your design in full-debug or partial-debug mode to get maximum performance in debug mode. For more information, refer to the section “[Compiling the Design in Debug Mode](#)”.

Using vcs

The syntax to use `vcs` is shown below:

```
% vcs [compile_options] Verilog_files
```

Commonly Used Options

This section lists some of the commonly used vcs options. For a complete list of options, see Appendix B, "Compile-time Options".

Options for Help and Documentation

-h or -help

Lists descriptions of the most commonly used VCS compile and runtime options.

-doc

Displays the VCS documentation in your system's default web browser.

-ID

Returns useful information such as VCS version and build date, VCS compiler version (same as VCS), and your work station's name, platform, and host ID (used in licensing).

Options for Licensing

+vcs+lic+wait [min]

Specifies minimum time for VCS to wait for a license.

Options for Accessing Verilog Libraries

-v *filename*

Specifies a Verilog library file. VCS looks in this file for definitions of the module and UDP instances that VCS found in your source code but for which it did not find the corresponding module or UDP definitions in your source code.

-y directory

Specifies a Verilog library directory. VCS looks in the source files in this directory for definitions of the module and UDP instances that VCS found in your source code but for which it did not find the corresponding module or UDP definitions in your source code. VCS looks in this directory for a file with the same name as the module or UDP identifier in the instance (not the instance name). If it finds this file, VCS looks in the file for the module or UDP definition to resolve the instance.

Note:

If you have multiple modules with the same name in different libraries, VCS selects the module defined in the library that is specified with the first -y option.

For example:

If rev1/cell.v and rev2/cell.v and rev3/cell.v all exist and define the module cell(), and you issue the following command:

```
% vcs -y rev1 -y rev2 -y rev3 +libext+.v top.v
```

VCS selects cell.v from rev1.

However, if the top.v file has a `uselib compiler directive as shown below:

```
//top.v
`uselib directory = /proj/libraries/rev3
//rest of top module code
//end top.v
```

...then `uselib takes priority. In this case, VCS will use rev3 / cell.v when you issue the following command:

```
% vcs -y rev1 -y rev2 +libext+.v top.v
```

Include the +libext compile-time option to specify the file name extension of the files you want VCS to look for in these directories.

+incdir+directory+

Specifies the directory or directories that VCS searches for include files used in the `include compiler directive. More than one directory may be specified when separated by the plus (+) character.

+libext+extension+

Specifies that VCS search only for files with the specified file name extensions in a library directory. You can specify more than one extension, separating the extensions with the plus (+) character. For example, +libext+.v+.V+ specifies searching for files with either the .v or .V extension in a library. The order in which you add file name extensions to this option does not specify an order in which VCS searches files in the library with these file name extensions.

+liborder

Specifies searching for module definitions for unresolved module instances through the remainder of the library where VCS finds the instance, then searching the next and then the next library on the vcs command line before searching in the first library on the command line.

Options for 64-bit Compilation

-full64

Enables compilation and simulation in 64-bit mode.

Option to Specify Files and Compile-time Options in a File

-file *filename*

Specifies a file containing a list of files and compile-time options.

Options for Debugging

-debug_pp

Enables minimal debug access to allow VPD dumping and assertion debug. You can view the results using DVE in post-processing mode.

Creates a VPD file (by using the \$vcapluson system task) and enables DVE for post-processing a design. Using -debug_pp can save compilation time by eliminating the overhead of compiling with -debug and -debug_all.

-debug

Enables interactive debugging using DVE or UCLI. You can force/release values on signals/wires/regs at runtime. It also enables everything that -debug_pp offers. However, line stepping is not enabled in this mode.

-debug_all

Enables full debugging using DVE or UCLI. You can enable line stepping, force/release any signal/reg/wire, etc. It also enables all other features that -debug_pp and -debug offers.

For information on DVE, refer to the “[Using Discovery Visual Environment](#)” chapter.

For information on UCLI, refer to the “[Using Unified Command-line Interface](#)” chapter.

Options for Discovery Visual Environment and UCLI

-gui

When used at compile time, always starts DVE at runtime.

-assert dve

Enables SystemVerilog assertions tracing in the VPD file. This tracing enables you to see assertion attempts.

For information on DVE, refer to the “[Using Discovery Visual Environment](#)” chapter.

For information on UCLI, refer to the “[Using Unified Command-line Interface](#)” chapter.

Options for Starting Simulation Right After Compilation

-R

Runs the executable file immediately after VCSlinks it together.

Options for Changing Parameter Values

-pvalue+parameter_hierarchical_name=value

Changes the specified parameter to the specified value. See

`-parameters filename`

Changes parameters specified in the file to values specified in the file. The syntax for a line in the file is as follows:

`assign value path_to_parameter`

The path to the parameter is similar to a hierarchical name except that you use the forward slash character (/) instead of a period as the delimiter.

For more information on overriding parameters, see “[Overriding Parameters](#)”.

Options for Controlling Messages

`-notice`

Enables verbose diagnostic messages.

`-q`

Quiet mode; suppresses messages such as those about the C compiler VCS is using, the source files VCS is parsing, the top-level modules, or the specified timescale.

`-V`

Verbose mode; compiles verbosely. The compiler driver program prints the commands it executes as it runs the C compiler, assembler, and linker.

Specifying a Log File

`-l filename`

Specifies a file where VCS records compilation messages. If you also enter the `-R` option, VCS records messages from both compilation and simulation in the same file.

Defining a Text Macro

`+define+macro=value+`

Defines a text macro in your source code to a value or character string. You can test for this definition in your Verilog source code using the `'ifdef` compiler directive. If there are blank spaces in the character string then you must enclose it in quotation marks.

For example:

```
vcs design.v +define+USELIB="dir=dir1 dir=dir2"
```

The macro is used in a `'uselib` compiler directive:

```
'uselib 'USELIB libext+.v
```

Specifying the Name of the Executable File

`-o name`

Specifies the name of the executable file. In UNIX, the default is `simv`.

Options For Coverage Metrics

`-cm line|cond|fsm|tgl|path|branch|assert`

Specifies compiling for the specified type or types of coverage.

The argument specifies the types of coverage:

line

Compile for line or statement coverage.

cond

Compile for condition coverage.

fsm

Compile for FSM coverage.

tgl

Compile for toggle coverage.

path

Compile for path coverage.

branch

Compile for branch coverage

assert

Compile for SystemVerilog assertion coverage.

For more information on Coverage, see “[Coverage](#)”.

Simulation

During compilation, using the intermediate files generated, VCS creates a binary executable, simv. You can use simv to run the simulation. Based on how you compile the design, you can run your simulation the following ways:

- Interactive mode
- Batch mode

For information on compiling the design, refer to the “[Compilation](#)” section.

Interactive Mode

You compile your design in interactive mode, also called debug mode, in the initial phase of your design cycle. In this phase, you require abilities to debug the design issues using a GUI or through the command line. To debug using a GUI, you can use the Discovery Verification Environment (DVE), and to debug through the command-line interface, you can use the Unified Command-line Interface (UCLI).

Note:

To simulate the design in the interactive mode, you must compile the design using the `-debug` or `-debug_all` compile-time options. For information on compiling the design, refer to the “[Compilation](#)” section.

Batch Mode

You compile your design in batch mode, also called as optimized mode, when most of your design issues are resolved. In this phase, you will be more interested to achieve better performance to run regressions, and with minimum debug abilities.

Note:

The runtime performance reduces if you use `-debug` or `-debug_all`. Use these options only when you require runtime debug abilities.

The following command line simulates the design in batch mode:

```
% simv
```

Commonly Used Runtime Options

Use the following command line to simulate the design:

```
% executable [runtime_options]
```

By default, VCS generates the binary executable `simv`. However, you can use the compile-time option, `-o` with the `vcs` command line to generate the binary executable with the specified name.

This section lists some of the commonly used runtime options. For a complete list of options, see “[Simulation Options](#)”.

Options for Coverage Metrics

`-cm line|cond|fsm|tgl|path|branch|assert`

Specifies monitoring for the specified type or types of coverage.
The arguments specify the types of coverage:

`line`

Monitors for line or statement coverage.

`cond`

Monitors for condition coverage.

`fsm`

Monitors for FSM coverage.

`tgl`

Monitors for toggle coverage.

`path`

Monitors for path coverage.

`branch`

Monitors for branch coverage

`assert`

Monitors for SystemVerilog assertion coverage

For more information on Coverage, see “[Coverage](#)”.

Options for Discovery Visual Environment and UCLI

`-l log_filename`

Specifies a log file that contains the commands you entered and the responses from VCS and DVE.

`-gui`

Starts DVE and loads the design to run in interactive mode.

`-ucli`

Starts the UCLI debugger command line.

`-do filename`

Specifies a file containing UCLI commands. VCS executes these at the start of simulation.

For the complete list of UCLI commands, refer to the chapter, “[Using Unified Command-line Interface](#)”.

Options for Recording Output

`-l filename`

Specifies writing all messages from simulation to the specified file, as well as, displaying these messages on the standard output.

3

Modeling Your Design

Verilog coding style is the most important factor that affects the simulation performance of a design. How you write your design can make the difference between a fast error-free simulation, and one that suffers from race conditions and poor performance. This chapter describes some Verilog modeling techniques that will help your code designs simulate most efficiently with VCS.

This chapter includes the following topics:

- “[Avoiding Race Conditions](#)”
- “[Optimizing Testbenches for Debugging](#)”
- “[Avoiding the Debugging Problems From Port Coercion](#)”
- “[Creating Models That Simulate Faster](#)”
- “[Case Statement Behavior](#)”

- “Memory Size Limits in VCS”
 - “Using Sparse Memory Models”
 - “Obtaining Scope Information”
 - “Avoiding Circular Dependency”
 - “Designing With \$lsi_dumpports for Simulation and Test”
-

Avoiding Race Conditions

A race condition is defined as a coding style for which there is more than one correct result. Since the output of the race condition is unpredictable, it can cause unexpected problems during simulation. It is easy to accidentally code race conditions in Verilog. For example, in *Digital Design with Verilog HDL* by Sternheim, Singh, and Trivedi, at least two of the examples provided with the book (adder and cachemem) have race conditions. VCS provides some tools for race detection.

Some common race conditions and ways of avoiding them are described in the following sections.

Using and Setting a Value at the Same Time

In this example, the two parallel blocks have no guaranteed ordering, so it is ambiguous whether the \$display statement will be executed.

```
module race;
    reg a;
    initial begin
        a = 0;
        #10 a = 1;
    end
    initial begin
        #10 if (a) $display("may not print");
    end
endmodule
```

The solution is to delay the \$display statement with a #0 delay:

```
initial begin
    #10 if (a)
        #0 $display("may not print");
end
```

You can also move it to the next time step with a non-zero delay.

Setting a Value Twice at the Same Time

In this example, the race condition occurs at time 10 because no ordering is guaranteed between the two parallel initial blocks.

```
module race;
    reg r1;
    initial #10 r1 = 0;
    initial #10 r1 = 1;
    initial
        #20 if (r1) $display("may not print");
endmodule
```

The solution is to stagger the assignments to register `r1` by finite time, so that the ordering of the assignments is guaranteed. Note that using the nonblocking assignment (`<=`) in both assignments to `r1` would not remove the race condition in this example.

Flip-Flop Race Condition

It is very common to have race conditions near latches or flip-flops. Here is one variant in which an intermediate node *a* between two flip-flops is set and sampled at the same time:

```
module test(out,in,clk);
    input in,clk;
    output out;
    wire a;
    dff dff0(a,in,clk);
    dff dff1(out,a,clk);
endmodule
module dff(q,d,clk);
    output q;
    input d,clk;
    reg q;
    always @(posedge clk)
        q = d;      // race!
endmodule
```

The solution for this case is straightforward. Use the nonblocking assignment in the flip-flop to guarantee the order of assignments to the output of the instances of the flip-flop and sampling of that output. The change looks like this:

```
always @(posedge clk)
    q <= d;      // ok
```

Or add a nonzero delay on the output of the flip-flop:

```
always @(posedge clk)
    q = #1 d;      // ok
```

Or use a nonzero delay in addition to the nonblocking form:

```
always @ (posedge clk)
    q <= #1 d;      // ok
```

Note that the following change does not resolve the race condition:

```
always @ (posedge clk)
    #1 q = d;      // race!
```

The `#1` delay simply shifts the original race by one time unit, so that the intermediate node is set and sampled one time unit *after* the posedge of clock, rather than *on the* posedge of clock. Avoid this coding style.

Continuous Assignment Evaluation

Continuous assignments with no delay are sometimes propagated earlier in VCS than in Verilog-XL. This is fully correct behavior, but exposes race conditions such as the one in the following code fragment:

```
assign x = y;
initial begin
    y = 1;
    #1
    y = 0;
    $display(x);
end
```

In VCS, this displays 0, while in Verilog-XL, it displays 1, because the assignment of the value to `x` races with the usage of that value by the `$display`.

Another example of this type of race condition is the following:

```
assign state0 = (state == 3'h0);
always @(posedge clk)
begin
    state = 0;
    if (state0)
        // do something
end
```

The modification of `state` may propagate to `state0` before the `if` statement, causing unexpected behavior. You can avoid this by using the nonblocking assignment to `state` in the procedural code as follows:

```
state <= 0;
if (state0)
    // do something
```

This guarantees that `state` is not updated until the end of the time step, that is, after the `if` statement has executed.

Counting Events

A different type of race condition occurs when code depends on the number of times events are triggered in the same time step. For instance, in the following example, if `A` and `B` change at the same time, it is unpredictable whether `count` is incremented once or twice:

```
always @(A or B)
count = count + 1;
```

Another form of this race condition is to toggle a register within the always block. If toggled once or twice, the result may be unexpected behavior.

The solution to this race condition is to make the code inside the always block insensitive to the number of times it is called.

Time Zero Race Conditions

The following race condition is subtle but very common:

```
always @(posedge clock)
    $display("May or may not display");
initial begin
    clock = 1;
    forever #50 clock = ~clock;
end
```

This is a race condition because the transition of clock to 1 (posedge) may happen before or after the event trigger (always @ (posedge clock)) is established. Often the race is not evident in the simulation result because reset occurs at time zero.

The solution to this race condition is to guarantee that no transitions take place at time zero of any signals inside event triggers. Rewrite the clock driver in the above example as follows:

```
initial begin
    clock = 1'bx;
    #50 clock = 1'b0;
    forever #50 clock = ~clock;
end
```

Optimizing Testbenches for Debugging

Testbenches typically execute debugging features, for example, displaying text in certain situations as specified with the \$monitor or \$display system tasks. Another debugging feature, which is typically enabled in testbenches, is writing simulation history files during simulation so that you can view the results after simulation. Among other things, these simulation history files record the simulation times at which the signals in your design change value. These simulation history files can be either ASCII Value-Change-Dump (VCD) files that you can input into a number of third-party viewers, or binary VPD files that you can input into DVE. The \$dumpvars system task specifies writing a VCD file and the \$vcdpluson system task specifies writing a VPD file. You can also input a VCD file to DVE, which translates the VCD file to a VPD file and then displays the results from the new VPD file. For details on using DVE, see the *Discovery Visual Environment User Guide*.

Debugging features significantly slow down the simulation performance of any logic simulator including VCS. This is particularly true for operations that make VCS display text on the screen and even more so for operations that make VCS write information to a file. For this reason, you'll want to be selective about where in your design and where in the development cycle of your design you enable debugging features. The following sections describe a number of techniques that you can use to choose when debugging features are enabled.

Conditional Compilation

Use `ifdef, `else, and `endif compiler directives in your testbench to specify which system tasks you want to compile for debugging features. Then, when you compile the design with the +define compile-time option on the command line (or when the `define compiler directive appears in the source code), VCS will compile these tasks for debugging features. For example:

```
initial
begin
`ifdef postprocess
$vcpluspluson(0,design_1);
$vcplusplustraceon(design_1);
$vcplusplusdeltacycleon;
$vcplusplusglitchon;
`endif
end
```

In this case, the vcs command is as follows:

```
% vcs testbench.v design.v +define+postprocess
```

The system tasks in this initial block record several types of information in a VPD file. You can use the VPD file with DVE to post-process the design. In this particular case, the information is for all the signals in the design, so the performance cost is extensive. You would only want to do this early in the development cycle of the design when finding bugs is more important than simulation speed.

The command line includes the +define+postprocess compile-time option, which tells VCS to compile the design with these system tasks compiled into the testbench.

Later in the development cycle of the design, you can compile the design without the `+define+postprocess` compile-time option and VCS will not compile these system tasks into the testbench. Doing so enables VCS to simulate your design much faster.

Advantages and Disadvantages

The advantage of this technique is that simulation can run faster than if you enable debugging features at runtime. When you use conditional compilation, VCS has all the information it needs at compile-time.

The disadvantage of this technique is that you have to recompile the testbench to include these system tasks in the testbench, thus increasing the overall compilation time in the development cycle of your design.

Synopsys recommends that you consider this technique as a way to prevent these system tasks from inadvertently remaining compiled into the testbench, later in the development cycle, when you want faster performance.

Enabling Debugging Features At Runtime

Use the `$test$plusargs` system function in place of the `'ifdef` compiler directives. The `$test$plusargs` system function checks for a plusarg runtime option on the `simv` command line.

Note:

A plusarg option is an option that has a plus (+) symbol as a prefix.

An example of the `$test$plusargs` system function is as follows:

```
initial
if ($test$plusargs("postprocess"))
begin
$vcddpluson(0,design_1);
$vcddplustraceon(design_1);
$vcddplusdeltacycleon;
$vcddplusglitchon;
end
```

In this technique you do not include the `+define` compile-time argument on the `vcs` command line. Instead you compile the system tasks into the testbench and then enable the execution of the system tasks with the runtime argument to the `$test$plusargs` system function. Therefore, in this example, the `simv` command line is as follows:

```
% simv +postprocess
```

During simulation VCS writes the VPD file with all the information specified by these system tasks. Later you can execute another `simv` command line, without the `+postprocess` runtime option. As a result, VCS does not write the VPD file, and therefore runs faster.

There is a pitfall to this technique. This system function will match any plusarg that has the function's argument as a prefix. For example:

```
module top;
initial
begin
if ( $test$plusargs("a") )
    $display("\n<<< Now a >>>\n");
else if ( $test$plusargs("ab") )
    $display("\n<<< Now ab >>>\n");
else if ( $test$plusargs("abc") )
    $display("\n<<< Now abc >>>\n");
end
endmodule
```

No matter whether you enter the +a, +ab, or +abc plusarg, when you simulate the executable, VCS always displays the following:

```
<<< Now a >>>
```

To avoid this pitfall, enter the longest plusarg first. For example, you would revise the previous example as follows:

```
module top;
initial
begin
if ( $test$plusargs("abc") )
    $display("\n<<< Now abc >>>\n");
else if ( $test$plusargs("ab") )
    $display("\n<<< Now ab >>>\n");
else if ( $test$plusargs("a") )
    $display("\n<<< Now a >>>\n");
end
endmodule
```

Advantages and Disadvantages

The advantage to using this technique is that you do not have to recompile the testbench in order to stop VCS from writing the VPD file. This technique is something to consider using, particularly early in the development cycle of your design, when you are fixing a lot of bugs and already doing a lot of recompilation.

The disadvantages to this technique are considerable. Compiling these system tasks, or any system tasks that write to a file, into the testbench requires VCS to compile the `simv` executable so that it is possible for it to write the VPD file when the runtime option is included on the command line. This means that the simulation runs significantly slower than if you don't compile these system tasks into the testbench. This impact on performance remains even when you don't include the runtime option on the `simv` command line.

Using the `$test$plusargs` system function forces VCS to consider the worst case scenario — plusargs will be used at runtime — and VCS generates the `simv` executable with the corresponding overhead to prepare for these plusargs. The more fixed information VCS has at compile-time, the more VCS can optimize `simv` for efficient simulation. Alternatively, the more user control at runtime, the more overhead VCS has to add to `simv` to accept runtime options, and the less efficient the simulation.

For this reason Synopsys recommends that if you use this technique, you should plan to abandon it fairly early in the development cycle and switch to either the conditional compilation technique for writing simulation history files, or a combination of the two techniques.

Combining the Techniques

Some users find that they have the greatest amount of control over the advantages and disadvantages of these techniques when they combine them. Consider the following example:

```
`ifdef compostprocess
initial
  if ($test$plusargs ("runpostprocess" ) )
    begin
      $vcpluspluson(0,design_1);
      $vcplusplustraceon(design_1);
      $vcsplusdeltacycleon;
      $vcplusplusglitchon;
    end
`endif
```

In this instance, both the `+define+compostprocess` compile-time option and the `+runpostprocess` runtime option are required for VCS to write the VPD file. This technique allows you to avoid recompiling just to prevent VCS from writing the file during the next simulation and also provides you with a way to recompile the testbench, later in the development cycle, to exclude these system tasks without first editing the source code for the testbench.

Avoiding the Debugging Problems From Port Coercion

The first Verilog simulator had a port collapsing algorithm that removed ports so it could simulate faster. In this simulator, you could still refer to a collapsed port, but inside the simulator, the port did not exist.

VCS mimics port collapsing so that an old, but reusable design, now simulated with VCS, will have the same simulation results. For this reason the default behavior of VCS is to “coerce” all ports to inout ports.

This port coercion can, for example, result in a value propagating up the design hierarchy out of a port you declared to be an input port and unexpectedly driving the signal connected to this input port. Port coercion, therefore, can cause debugging problems.

Port coercion also results in slower simulation, because with port coercion VCS must be prepared for bidirectional behavior of input and output ports as well as inout ports.

To avoid these debugging problems, and to increase simulation performance, do the following when writing new models:

1. If you need values to propagate in and out of a port, declare it as an inout port. If you don’t need this bidirectional behavior, declare it as an input or output port.
2. Compile the modules with these ports under the `'noportcoerce` compiler directive.

Creating Models That Simulate Faster

When modeling your design, for faster simulation use higher levels of abstraction. Behavioral and RTL models simulate much faster than gate and switch level models. This rule of thumb is not unique to VCS; it applies to all Verilog simulators and even all logic simulators in general.

What is unique to VCS are the acceleration algorithms that make behavioral and RTL models simulate even faster. In fact, VCS is particularly optimized for RTL models for which simulation performance is critical.

These acceleration algorithms work better for some designs than for others. Certain types of designs prevent VCS from applying some of these algorithms. This section describes the design styles that simulate faster or slower.

The acceleration algorithms apply to most data types and primitives and most types of statements but not all of them. This section also describes the data types, primitives, and types of statements that you should try to avoid.

VCS is optimized for simulating sequential devices. Under certain circumstances VCS infers that an `always` block is a sequential device and simulates the `always` block much faster. This section describes the coding guidelines you should follow to make VCS infer an `always` block as a sequential device.

When writing an `always` block, if you cannot follow the inferencing rules for a sequential device there are still things that you should keep in mind so that VCS simulates the `always` block faster. This section also describes the guidelines for coding faster simulating `always` blocks that VCS infers to be combinatorial instead of sequential devices.

Unaccelerated Data Types, Primitives, and Statements

VCS cannot accelerate certain data types and primitives. VCS also cannot accelerate certain types of statements. This section describes the data types, primitives, and types of statements that you should try to avoid.

Avoid Unaccelerated Data Types

VCS cannot accelerate certain data types. The following table lists these data types:

Data Type	Description in IEEE Std 1364-2001
time and realtime	Page 22
real	Page 22
named event	Page 138
trireg net	Page 26
integer array	Page 22

Avoid Unaccelerated Primitives

VCS cannot accelerate tranif1, tranif0, rtranif1, rtranif0, tran, and rtran switches. They are defined in IEEE Std 1364-2001 page 86.

Avoid Calls to User-defined Tasks or Functions Declared in Another Module

VCS cannot accelerate user-defined tasks or functions declared in another module. For example:

```
module bottom (x,y);  
.  
.  
.  
always @ y  
top.task_identifier(y,rb);  
endmodule
```

Avoid Strength Specifications in Continuous Assignment Statements

Omit strength specifications in continuous assignment statements. For example:

```
assign net1 = flag1;
```

Simulates faster than:

```
assign (strong1, pullo) net1= flag1;
```

Continuous assignment statements are described on IEEE 1364-2001 pages 69-70.

Inferring Faster Simulating Sequential Devices

VCS is optimized to simulate sequential devices. If VCS can infer that an always block behaves like a sequential device, VCS can simulate the always block much faster.

The IEEE Std 1364-2001 defines `always` constructs on page 149. Verilog users commonly use the term `always` block when referring to an `always` construct.

VCS can infer whether an `always` block is a combinatorial or sequential device. This section describes the basis on which VCS makes this inference.

Avoid Unaccelerated Statements

VCS does not infer an `always` block to be a sequential device if it contains any of the following statements:

Statement	Description in IEEE Std 1364-2001
<code>force</code> and <code>release</code> procedural statements	Page 126-127
<code>repeat</code> statements	Page 134-135, see the other looping statements on these pages and consider them as an alternative.
<code>wait</code> statements, also called level-sensitive event controls	Page 141
<code>disable</code> statements	Page 162-164
<code>fork-join</code> block statements, also called parallel blocks	Page 146-147

Using either blocking or nonblocking procedural assignment statements in the `always` block does not prevent VCS from inferring a sequential device, but in VCS blocking procedural assignment statements are more efficient.

Synopsys recommends zero delay nonblocking assignment statements to avoid race conditions.

IEEE Std 1364-2001 describes blocking and nonblocking procedural assignment statements on pages 119-124.

Place Task Enabling Statements in Their Own always Block and Use No Delays

IEEE Std 1364-2001 defines tasks and task enabling statements on pages 151-156.

VCS infers that an `always` block that contains a task enabling statement is a sequential device only when there are no delays in the task declaration.

All Sequential Controls Must Be in the Sensitivity List

To borrow a concept from VHDL, the sensitivity list for an `always` block is the event control that immediately follows the `always` keyword.

IEEE Std 1364-2001 defines event controls on page 138 and mentions sensitivity lists on page 139.

For correct inference, all sequential controls must be in the sensitivity list. The following code examples illustrate this rule:

- VCS does not infer the following DFF to be a sequential device:

```
always @ (d)
  @ (posedge clk) q <=d;
```

Even though `clk` is in an event control, it is not in the sensitivity list event control.

- VCS does not infer the following latch to be a sequential device:

```
always begin
    wait clk; q <= d; @ d;
end
```

There is no sensitivity list event control.

- VCS infers the following latch to be a sequential device:

```
always @ (clk or d)
    if (clk) q <= d;
```

The sequential controls, `clk` and `d`, are in the sensitivity list event control.

Avoid Level-sensitive Sensitivity Lists Whose Signals are Used “Completely”

VCS infers a combinational device instead of a sequential device if the following conditions are both met:

- The sensitivity list event control is level sensitive

A level sensitive event control does not contain the `posedge` or `negedge` keywords.

- The signals in the sensitivity list event control are used “completely” in the `always` block

Used “completely” means that there is a possible simulation event if the signal has a true or a false (`1` or `0`) value.

The following code examples illustrate this rule:

Example 1

VCS infers that the following `always` block is combinatorial, not sequential:

```
always @ (a or b)
    y = a or b
```

Here the sensitivity list event control is level sensitive and VCS assigns a value to `y` whether `a` or `b` are true or false.

Example 2

VCS also infers that the following `always` block is combinatorial, not sequential:

```
always @ (sel or a or b)
    if (sel)
        y=a;
    else
        y=b;
```

Here the sensitivity list event control is also level sensitive and VCS assigns a value to `y` whether `a`, `b`, or `sel` are true or false. Note that the `if-else` conditional statement uses signal `sel` completely, VCS executes an assignment statement whether `sel` is true or false.

Example 3

VCS infers that the following `always` block is sequential:

```
always @ (sel or a or b)
    if (sel)
        y=a;
```

In this instance, there is no simulation event when signal `sel` is false (0).

Modeling Faster always Blocks

Whether VCS infers an `always` block to be a sequential device or not, there are modeling techniques you should use for faster simulation.

Place All Signals Being Read in the Sensitivity List

The sensitivity list for an `always` block is the event control that immediately follows the `always` keyword. Place all nets and registers, whose values you are assigning to other registers, in the `always` block, and place all nets and registers, whose value changes trigger simulation events, in the sensitivity list control.

Use Blocking Procedural Assignment Statements

In VCS, blocking procedural assignment statements are more efficient.

Synopsys recommends zero delay nonblocking procedural assignment statements to avoid race conditions.

IEEE Std 1364-2001 describes blocking and nonblocking procedural assignment statements on pages 119-124.

Avoid force and release Procedural Statements

IEEE Std 1364-2001 defines these statements on pages 126-127. A few occurrences of these statements in combinatorial `always` blocks does not noticeably slow down simulation but their frequent use does lead to a performance cost.

Using the +v2k Compile-Time Option

The following table lists the implemented constructs in Std 1364-2001 and whether you need the `+v2k` compile-time option to use them.

Std 1364-2001 Construct	Require +v2k
comma separated event control expressions: <code>always @ (r1, r2, r3)</code>	yes
name-based parameter passing: <code>modname #(.param_name (value)) inst_name (sig1, ...);</code>	yes
ANSI-style port and argument lists: <code>module dev (output reg [7:0] out1, input wire [7:0] w1);</code>	yes
initialize a reg in its declaration: <code>reg [15:0] r2 = 0;</code>	yes
conditional compiler directives: <code>'ifndef and 'elseif</code>	yes
disabling the default net data type: <code>'default_nettype</code>	yes
signed arithmetic extensions: <code>reg signed [7:0] r1;</code>	no
file I/O system tasks: <code>\$fopen \$fsanf \$scanf and more</code>	no
passing values from the runtime command line: <code>\$value\$plusarg system function</code>	yes
indexed part-selects: <code>reg1 [8+:5]=5'b11111;</code>	yes
multi-dimensional arrays: <code>reg [7:0] r1 [3:0] [3:0];</code>	yes
maintaining file name and line number: <code>'line</code>	yes
implicit event control expression lists: <code>always @*</code>	yes

Std 1364-2001 Construct	Require +v2k
the power operator: r1=r2**r3;	yes
attributes: (* optimize_power=1 *)	yes
module dev (res,out,clk,data1,data2);	
generate statements	yes
localparam declarations	yes
Automatic tasks and functions task automatic t1();	requires the -sverilog compile-time option
constant functions	yes
localparam lp1 = const_func(p1);	
parameters with a bit range parameter bit [7:0] [31:0] P = {32'd1,32'd2,32'd3,32'd4,32'd5,32'd6,32'd7,32'd8};	requires the -sverilog compile-time option

Case Statement Behavior

The IEEE Std 1364-2001 standards for the Verilog language state that you can enter the question mark character (?) in place of the z character in casex and casez statements. The standard does not specify that you can also make this substitution in case statements and you might infer that this substitution is not allowed in case statements.

VCS, like other Verilog simulators, does not make this inference, and allows you to also substitute ? for z in case statements. If you do, remember that z does not stand for "don't care" in a case statement, like it does in a casez or casex statement. In a case statement z stands for the usual high impedance and therefore so does ?.

Precedence in Text Macro Definitions

In text macros, the line continuation character (\) has a higher precedence than the one line comment characters (//). This means that VCS can merge a subsequent line with the text in a one line comment, for example:

```
`define print_me_1 \
$display( "Hello 1" ); // just a comment \
$display( "I'm OK" );
```

VCS merges the second \$display system task with the comment on the previous line and does not display the text string I 'm OK.

Memory Size Limits in VCS

The bit width for a word or an element in a memory in VCS must be less than 0x100000 (or 2^{20} or 1,048,576) bits.

The number of elements or words (sometimes also called rows) in a memory in VCS must be less than 0x3FFF_FFFE-1 (or $2^{30} - 2$ or 1,073,741,822) elements or words.

The total bit count of a memory (total number of elements * word size) must be less than $8 * (1024 * 1024 * 1024 - 2)$ or 8,573,157,376.

Using Sparse Memory Models

If your design contains a large memory, the `simv` executable will need large amounts of machine memory to simulate it. However, if your simulation only accesses a small number of elements in the design's memory, you can use a sparse memory model to significantly reduce the amount of machine memory that VCS will need to simulate your design.

You use the `/*sparse*/` pragma or metacomment in the memory declaration to specify a sparse memory model. For example:

```
reg /*sparse*/ [31:0] pattern [0:10_000_000];
integer i, j;
initial
begin
    for (j=1; j<10_000; j=j+1)
        for (i=0; i<10_000_000; i=i+1_000)
            pattern[i] = i+j;
end
endmodule
```

In simulations, this memory model uses 4 MB of machine memory with the `/*sparse*/` pragma, 81 MB without it. There is a small runtime performance cost to sparse memory models: the simulation of the memory with the `/*sparse*/` pragma took 64 seconds, 56 seconds without it.

The larger the memory, and the fewer elements in the memory that your design reads or writes to, the more machine memory you will save by using this feature. It is intended for memories that contain at least a few MBs. If your design accesses 1% of its elements you could save 97% of machine memory. If your design accesses 50% of its elements, you save 25% of machine memory. Do not use this

feature if your design accesses more than 50% of its elements because using the feature in these cases may lead to more memory consumption than not using it.

Using sparse memory models does not increase the memory size limits described in the previous section.

Note:

- Sparse memory models cannot be manipulated by PLI applications through `tf` calls (the `tf_nodeinfo` routine issues a warning for sparse memory and returns NULL for the memory handle).
- Sparse memory models cannot be used as a personality matrix in PLA system tasks.

Obtaining Scope Information

VCS has custom format specifications (IEEE Std 1364-2001 does not define these) for displaying scope information. It also has system functions for returning information about the current scope.

Scope Format Specifications

The IEEE Std 1364-2001 describes the `%m` format specification for system tasks for displaying information such as `$write` and `$display`. The `%m` specification tells VCS to display the hierarchical name of the module instance that contains the system task. If the system task is in a scope lower than a module instance, it tells VCS to do the following:

- In named begin-end or fork-join blocks, it adds the block name to the hierarchical name.
- In user-defined tasks or functions, it considers the hierarchical name of the task declaration or function definition as the hierarchical name of the module instance.

VCS has the following additional format specifications for displaying scope information:

`%i`

Specifies the same as `%m` with the following difference: when in a user-defined task or function, the hierarchical name is the hierarchical name of the instance or named block containing the task enabling statement or function call, not the hierarchical name of the task or function declaration.

If the task enabling statement is in another user-defined task, the hierarchical name is the hierarchical name of the instance or named block containing the task enabling statement for this other user-defined task.

If the function call is in another user-defined function, the hierarchical name is the hierarchical name of the instance or named block containing the function call for this other user-defined function.

If the function call is in a user-defined task, the hierarchical name is the hierarchical name of the instance or named block containing the task enabling statement for this user-defined task.

`%-i`

Specifies that the hierarchical name is always of a module instance, not a named block or user-defined task or function. If the system task (such as `$write` and `$display`) is in:

- A named block — the hierarchical name is that of the module instance that contains the named block
- A user-defined task or function — the hierarchical name is that of the module instance containing the task enabling statement or function call

Note:

The `%i` and `%-i` format specifications are not supported with the `$monitor` system task.

The following commented code example shows what these format specifications do:

```
module top;
reg r1;

task my_task;
input taskin;
begin
$display("%m");           // displays "top.my_task"
$display("%i");           // displays "top.d1.named"
$display("%-i");          // displays "top.d1"
end
endtask

function my_func;
input taskin;
begin
$display("%m");           // displays "top.my_func"
$display("%i");           // displays "top.d1.named"
$display("%-i");          // displays "top.d1"
end
endfunction

dev1 d1 (r1);
endmodule
```

```

module dev1(input);
  input inport;

  initial
  begin:named
    reg namedreg;
    $display("%m"); // displays "top.d1.named"
    $display("%i"); // displays "top.d1.named"
    $display("%-i"); // displays "top.d1"
    namedreg=1;
    top.my_task(namedreg);
    namedreg = top.my_func(namedreg);
  end

endmodule

```

Returning Information About the Scope

The `$activeinst` system function returns information about the module instance that contains this system function. The `$activescope` system function returns information about the scope that contains the system function. This scope can be a module instance, a named block, a user-defined task, or a function in a module instance.

When VCS executes these system functions, it performs the following:

1. Stores the current scope in a temporary location.

2. If there are no arguments, it returns a pointer to the temporary location. Pointers are not used in Verilog but they are in DirectC applications.

The possible arguments are hierarchical names. If there are arguments, it compares them from left to right with the current scope. If an argument matches, the system function returns a 32-bit non-zero value. If none of the arguments match the current scope, the system function returns a 32-bit zero value.

The following example contains these system functions:

```
module top;
reg r1;
initial
r1=1;
dev1 d1(r1);
endmodule

module dev1(in);
input in;
always @ (posedge in)
begin:named
if ($activeinst("top.d0","top.d1"))
$display("%i");
if ($activescope("top.d0.block","top.d1.named"))
$display("%-i");
end
endmodule
```

The following is an example of a DirectC application that uses the \$activeinst system function:

```
extern void showInst(input bit[31:0]);  
  
module discriminator;  
task t;  
reg[31:0] r;  
begin  
    showInst($activeinst);  
    if($activeinst("top.c1", "top.c3"))  
    begin  
        r = $activeinst;  
        $display("for instance %i the pointer is %s", r ? "non-zero" : "zero");  
    end  
end  
endtask  
  
module child;  
initial discriminator.t;  
endmodule  
  
module top;  
child c1();  
child c2();  
child c3();  
child c4();  
endmodule
```

declaration of C function named showInst

\$activeinst system function without arguments passed to the C function

In task t, the following occurs:

1. The \$activeinst system function returns a pointer to the current scope, which is passed to the C function showInst. It is a pointer to a volatile or temporary char buffer containing the name of the instance.
2. A nested begin block executes only if the current scope is either top.c1 or top.c3.
3. VCS displays whether \$activeinst points to a zero or non-zero value.

The C code is as follows:

```
#include <stdio.h>

void showInst(unsigned str_arg)
{
    const char *str = (const char *)str_arg;
    printf("DirectC: [%s]\n", str);
}
```

Function `showInst` declares the `char` pointer `str` and assigns to it the value of its parameter, which is the pointer in `$activeinst` in the Verilog code. Then with a `printf` statement, it displays the hierarchical name that `str` is pointing to. Notice that the function begins the information it displays with `DirectC:` so that you can differentiate it from what VCS displays.

During simulation VCS and the C function display the following:

```
DirectC: [top.c1]
for instance top.c1 the pointer is non-zero
DirectC: [top.c2]
DirectC: [top.c3]
for instance top.c3 the pointer is non-zero
DirectC: [top.c4]
```

Avoiding Circular Dependency

The \$random system function has an optional seed argument. You can use this argument to make the return value of this system function the assigned value in a continuous assignment, procedural continuous assignment, or force statement. For example:

```
assign out = $random(in);  
  
initial  
begin  
  assign dr1 = $random(in);  
  force dr2 = $random(in);
```

When you do this, you might set up a circular dependency between the seed value and the statement, resulting in an infinite loop and a simulation failure.

This circular dependency doesn't usually occur, but it can occur, so VCS displays a warning message when you use a seeded argument with these kinds of statements. This warning message is as follows:

```
Warning-[RWSI] $random() with a 'seed' input  
$random in the following statement was called with a 'seed' input  
This may cause an infinite loop and an eventual crash at runtime.  
"expl.v", 24: assign dr1 = $random(in);
```

The warning message ends with the source file name and line number of the statement, followed by the statement itself.

This possible circular dependency does not occur either when you use a seed argument and the return value is the assigned value in a procedural assignment statement, or when you do not use the seed argument in a continuous, procedural continuous, or force statement.

For example:

```
assign out = $random();  
  
initial  
begin  
    assign dr1 = $random();  
    force dr2 = $random();  
    dr3 = $random(in);
```

These statements do not generate the warning message.

You can tell VCS not to display the warning message by using the `+warn=noRWSI` compile-time argument and option.

Designing With `$lsi_dumpports` for Simulation and Test

This section is intended to provide guidance when using `$lsi_dumpports` with Automatic Test Pattern Generation (ATPG) tools. Occasionally, ATPG tools strictly follow port direction and do not allow unidirectional ports to be driven from within the device. If you are not careful while writing the test fixture, the results of `$lsi_dumpports` causes problems for ATPG tools.

Note:

See “[Signal Value/Strength Codes](#)”. These are based on the TSSI Standard Events Format State Character set.

Dealing With Unassigned Nets

Consider the following example:

```
module test(A);
  input A;
  wire A;
  DUT DUT_1 (A);
  // assign A = 1'bz;
  initial
    $lsi_dumpports(DUT_1, "dump.out");
  endmodule

module DUT(A);
  input A;
  wire A;
  child child_1(A);
endmodule

module child(A);
  input A;
  wire Z,A,B;
  and (Z,A,B);
endmodule
```

In this case, the top-level wire `A` is undriven at the top level. It is an input which goes to an input in `DUT_1`, then to an input in `CHILD_1` and finally to an input of an AND gate in `CHILD_1`. When `$lsi_dumpports` evaluates the drivers on port `A` of `test.DUT_1`, it finds no drivers on either side of port `A` of `DUT_1`, and therefore gives a code of `F`, tristate (input and output unconnected).

The designer actually meant for a code of `Z` to be returned, input tristated. To achieve this code, the input `A` needs to be assigned a value of `z`. This is achieved by removing the comment from the line, `// assign A = 1'bz;`, in the above code. Now, when the code

is executed, VCS is able to identify that the wire A going into DUT_1 is being driven to a z. With the wire driven from the outside and not the inside, \$lsi_dumpports returns a code of z.

Code Values at Time 0

Another issue can occur at time 0, before values have been assigned to ports as you intended. As a result, \$lsi_dumpports makes an evaluation for drivers when all of the users intended assignments haven't been made. To correct this situation, you need to advance simulation time just enough to have your assignments take place. This can be accomplished by adding a #1 before \$lsi_dumpports as follows:

```
initial  
begin  
#1 $lsi_dumpports(instance, "dump.out");  
end
```

Cross Module Forces and No Instance Instantiation

In the following example there are two problems.

```
module test;
initial
begin
force top.u1.a = 1'b0;
$lsi_dumpports(top.u1, "dump.out");
end
endmodule

module top;
middle u1 (a);
endmodule

module middle(a);
input a;
wire b;
buf(b,a);
endmodule
```

First, there is no instance name specified for `$lsi_dumpports`. The syntax for `$lsi_dumpports` calls for an instance name. Since the user didn't instantiate module `top` in the test fixture, they are left specifying the MODULE name `top`. This will produce a warning message from VCS. Since `top` appears only once, that instance will be assumed.

The second problem comes from the cross-module reference (XMR) that the force command uses. Since the module `test` doesn't instantiate `top`, the example uses an XMR to force the desired signal. The signal being forced is port `a` in instance `u1`. The problem here is that this force is done on the port from within the instance `u1`. The user expects this port `a` of `u1` to be an input, but when

`$lsi_dumpports` evaluates the ports for the drivers, it finds that port `a` of instance `u1` is being driven from inside and therefore returns a code of `L`.

To correct these two problems, you need to instantiate `top` inside `test`, and drive the signal `a` from within `test`. This is done in the following way:

```
module test;
  wire a;
  initial
    begin
      force a = 1'b0;
      $lsi_dumpports(test.u0.u1, "dump.out2");
    end
    top u0 (a);
  endmodule

module top(a);
  input a;
  middle u1 (a);
endmodule

module middle(a);
  input a;
  wire b;
  buf(b,a);
endmodule
```

By using the method in this example, the port `a` of instance `u1` is driven from the outside, and when `$lsi_dumpports` checks for the drivers it reports a code of `D` as desired.

Signal Value/Strength Codes

The enhanced state character set is based on the TSSI Standard Events Format State Character set with additional expansion to include more unknown states. The supported character set is as follows:

Testbench Level (only z drivers from the DUT)

D	low
U	high
N	unknown
Z	tristate
d	low (2 or more test fixture drivers active)
u	high (2 or more test fixture drivers active)

DUT Level (only z drivers from the testbench)

L	low
H	high
X	unknown (don't care)
T	tristate
I	low (2 or more DUT drivers active)

Testbench Level (only z drivers from the DUT)

h	high (2 or more DUT drivers active)
---	-------------------------------------

Drivers Active on Both Levels

0	low (both input and output are active with 0 values)
1	high (both input and output are active with 1 values)
?	unknown
F	tristate (input and output unconnected)
A	unknown (input 0 and output unconnected)
a	unknown (input 0 and output X)
B	unknown (input 1 and output 0)
b	unknown (input 1 and output X)
C	unknown (input X and output 0)

c	unknown (input X and output 1)
f	unknown (input and output tristate)

4

Compiling the Design

This chapter describes the following sections:

- “[Compiling the Design in Debug Mode](#)”
- “[Compiling the Design in Optimized Mode](#)”
- “[Key Compilation Features](#)”

Compiling the Design in Debug Mode

Debug mode, also called interactive mode, is typically used (but not limited to):

- During your initial phase of the design, when you need to debug the design using debug tools like DVE, or UCLI.
- If you are using PLIs.

- If you use the UCLI commands to force a signal, to write into a registers/nets

VCS has the following compile-time options for debug mode:

- `-debug_pp` - Enables minimal debugging capabilities.

The `-debug_pp` option allows you to dump VCD, VPD, or EVCD files. You can also use this option to invoke DVE and UCLI. This option provides better runtime performance than the `-debug` or `-debug_all` options.

Note:

The `-debug_pp` option can only be used for post-processing. You cannot work with the executable in interactive mode with DVE or UCLI.

- `-debug` - Enables interactive debugging using DVE or UCLI. You can force/release values on signals/wires/regs at runtime. It also enables everything that `-debug_pp` offers. However, line stepping is not enabled in this mode.
- `-debug_all` - Enables full debugging using DVE or UCLI. You can enable line stepping, force/release any signal/reg/wire, etc. It also enables all other features that `-debug_pp` and `-debug` offer.

The following examples show how to compile the design in full and partial debug modes.

Compiling the design in partial debug mode

```
% vcs -debug [compile_options] TOP.v
```

Compiling the design in full debug mode

```
% vcs -debug_all [compile_options] TOP.v
```

For information on DVE, refer to chapter entitled, “[Using Discovery Visual Environment](#)”.

For information on UCLI, refer to the chapter entitled, “[Using Unified Command-line Interface](#)”.

Compiling the Design in Optimized Mode

Optimized mode is used when your design is fully-verified for design correctness, and is ready for regressions. VCS runtime performance is best in this mode when VCS optimizes a design.

For more information on performance, refer to the chapter entitled, [Chapter 10, "Performance Tuning"](#).

Note:

The runtime performance reduces if you use the `-debug` or `-debug_all` options. Use these options only when you require runtime debug capabilities.

Key Compilation Features

This section describes the following features in detail with a usage model and an example:

- “[Initializing Verilog Memories and Registers](#)”
- “[Overriding Parameters](#)”
- “[Checking for X and Z Values In Conditional Expressions](#)”
- “[VCS V2K Configurations and Libmaps](#)”

- “Using +evalorder Option”
-

Initializing Verilog Memories and Registers

VCS has compile-time options to initialize all bits of your Verilog memories and regs to the 0, 1, X, or Z value. These options are:

+vcs+initmem+0 | 1 | x | z

Initializes all bits of all memories in the design.

+vcs+initreg+0 | 1 | x | z

Initializes all bits of all regs in the design.

Note:

+vcs+initmem+, and +vcs+initreg+ options work only for the Verilog portion of the design.

The +vcs+initmem option initializes regular memories and multidimensional arrays of the reg data type. For example:

```
reg [7:0] mem [7:0] [15:0];
```

The +vcs+initmem option does not initialize multi-dimensional arrays of any other data type.

The +vcs+initreg option does not initialize registers (variables) other than the reg data type.

To prevent race conditions, avoid the following when you use these options:

- Assigning initial values to a regs in their declaration when the value you assign is not the same as the value specified with the `+vcs+initreg` option.

For example:

```
reg [7:0] r1 8'b01010101;
```

- Assigning values to regs or memory elements at simulation time 0 when the value you assign is not the same as the value specified with the `+vcs+initreg` or `+vcs+initmem` option.

For example:

```
initial  
begin  
mem[1] [1]=8'b00000001;
```

Usage Model

Compilation

```
% vcs [vcs_options] file1.v file2.v file3.v
```

Simulation

```
% simv [simv_options]
```

Overriding Parameters

There are two compile-time options for changing parameter values from the `vcs` command line:

- `-pvalue`

- -parameters

You specify a parameter with the -pvalue option. It has the following syntax:

```
vcs -pvalue+hierarchical_name_of_parameter=value
```

For example:

```
vcs source.v -pvalue+test.d1.param1=33
```

You specify a file with the -parameters option. The file contains command lines for changing values. A line in the file has the following syntax:

assign value path_to_the_parameter

Here:

assign

Keyword that starts a line in the file.

value

New value of the parameter.

path_to_the_parameter

Hierarchical path to the parameter. This entry is similar to a Verilog hierarchical name except that you use forward slash characters (/), instead of periods, as the delimiters.

The following is an example of the contents of this file:

```
assign 33 test/d1/param1  
assign 27 test/d1/param2
```

Note:

The `-parameters` and `-pvalue` options do not work with a `localparam` or a `specparam`.

Checking for X and Z Values In Conditional Expressions

The `-xzcheck` compile-time option tells VCS to display a warning message when it evaluates a conditional expression and finds it to have an X or Z value.

A conditional expression is of the following types or statements:

- A conditional or `if` statement:

```
if(conditional_exp)  
    $display("conditional_exp is true");
```

- A case statement:

```
case(conditional_exp)  
    1'b1: sig2=1;  
    1'b0: sig3=1;  
    1'bx: sig4=1;  
    1'bz: sig5=1;  
endcase
```

- A statement using the conditional operator:

```
reg1 = conditional_exp ? 1'b1 : 1'b0;
```

The following is an example of the warning message that VCS displays when it evaluates the conditional expression and finds it to have an X or Z value:

```
warning 'signal_name' within scope hier_name in file_name.v:  
line_number to x/z at time simulation_time
```

VCS displays this warning every time it evaluates the conditional expression to have an X or Z value, not just when the signal or signals in the expression transition to an X or Z value.

VCS does not display a warning message when a sub-expression has the value X or Z, but the conditional expression evaluates to a 1 or 0 value. For example:

```
r1 = 1'bz;  
r2 = 1'b1;  
if ( (r1 && r2) || 1'b1)  
    r3 = 1;
```

In this example, the conditional expression always evaluates to a value of 1. Therefore, VCS does not display a warning message.

Enabling the Checking

The `-xzcheck` compile-time option globally checks all the conditional expressions in the design and displays a warning message every time it evaluates a conditional expression to have an X or Z value. You can suppress or enable these warning messages on selected modules using `$xzcheckoff` and `$xzcheckon` system tasks. For more details on `$xzcheckoff` and `$xzcheckon` system tasks, see “[Checking for X and Z Values in Conditional Expressions](#)” on page D-43.

The `-xzcheck` compile-time option has an optional argument to suppress the warning for glitches evaluating to `X` or `Z` value. Synopsys calls these glitches as false negatives. See “[Filtering Out False Negatives](#)” on page 4-9.

Filtering Out False Negatives

By default, if a signal in a conditional expression transitions to an `X` or `Z` value and then to 0 or 1 in the same simulation time step, VCS displays the warning.

Example 1

In this example, VCS displays the warning message when reg `r1` transitions from 0 to `X` to 1 during simulation time 1.

Example 4-1 False Negative Example

```
module test;
reg r1;

initial
begin
r1=1'b0;
#1 r1=1'bx;
#0 r1=1'b1;
end

always @ (r1)
begin
if (r1)
$display("\n r1 true at %0t\n",$time);
else
$display("\n r1 false at %0t\n",$time);
end
endmodule
```

Example 2

In this example, VCS displays the warning message when reg r1 transitions from 1 to x during simulation time 1.

Example 4-2 False Negative Example

```
module test;
reg r1;

initial
begin
r1=1'b0;
#1 r1<=1'b1;
r1=1'bx;
end
always @ (r1)
begin
if (r1)
    $display("\n r1 true at %0t\n", $time);
else
    $display("\n r1 false at %0t\n", $time);
end

endmodule
```

If you consider these warning messages to be false negatives, use the `nofalseneg` argument to the `-xzcheck` option to suppress the messages.

For example:

```
% vcs -xzchecknofalseneg example.v
```

If you compile and simulate example1 or example2 with the `-xzcheck` compilation option, but without the `nofalseneg` argument, VCS displays the following warning about signal `r1` transitioning to an X or Z value:

```
r1 false at 0
Warning: 'r1' within scope test in source.v: 13 goes to x/
z at time 1

r1 false at 1

r1 true at 1
```

If you compile and simulate the examples shown earlier in this chapter, Example 1 or Example 2, with the `-xzcheck` compilation option and the `nofalseneg` argument, VCS does not display the warning message.

VCS V2K Configurations and Libmaps

Library mapping files are an alternative to the defacto standard way of specifying Verilog library directories and files with the `-v`, `-y`, and `+libext+ext` compile-time options and the `'uselib` compiler directive.

Configurations use the contents of library mapping files to specify what source code to use to resolve instances in other parts of your source code.

Library mapping and configurations are described in Std 1364-2001 IEEE Verilog Hardware Description Language. There is additional information on SystemVerilog in Std 1800-2005 IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language.

It specifies that SystemVerilog interfaces can be assigned to logical libraries.

Library Mapping Files

A library mapping file enables you to specify logical libraries and assign source files to these libraries. You can specify one or more logical libraries in the library mapping file. If you specify more than one logical library, you are also specifying the search order VCS uses to resolve instances in your design.

The following is an example of the contents of a library mapping file:

```
library lib1 /net/design1/design1_1/*.v;  
library lib2 /net/design1/design1_2/*.v;
```

Note:

Path names can be absolute or relative to the current directory that contains the library mapping file.

In this example library mapping file, there are two logical libraries. VCS searches the source code assigned to `lib1` first to resolve module instances (or user-defined primitive or SystemVerilog interface instances) because that logical library is listed first in the library mapping file.

When you use a library mapping file, source files that are not assigned to a logical library in this file are assigned to the default logical library named `work`.

You specify the library mapping file with the `-libmap` during compilation. Library mapping file is a Verilog 2001 feature, therefore, use `+v2k` or `-sverilog` along with `-libmap`.

Configurations

Verilog 2001 configurations are sets of rules that specify what source code is used for particular instances.

Verilog 2001 introduces the concept of configurations and it also introduces the concept of cells. A cell is like a VHDL design unit. A module definition is a type of cell, as is a user-defined primitive. Similarly, a configuration is also a cell. A SystemVerilog interface and testbench program block are also types of cells.

Configurations do the following:

- Specify a library search order for resolving cell instances (as does a library mapping file)
- Specifies overrides to the logical library search order for specified instances
- Specifies overrides to the logical library search order for all instances of specified cells

You can define a configuration in a library mapping file or in any type of Verilog source file outside the module definition.

Configurations can be mapped to a logical library just like any other type of cell.

Configuration Syntax

A configuration contains the following statements:

```
config config_identifier;  
design [library_identifier.]cell_identifier;  
config_rule_statement;  
endconfig
```

Where:

`config`

Is the keyword that begins a configuration.

`config_identifier`

Is the name you enter for the configuration.

`design`

Is the keyword that starts a `design` statement for specifying the top of the design.

`[library_identifier.]cell_identifier;`

Specifies the top-level module (or top-level modules) in the design and the logical library for this module (modules).

`config_rule_statement`

Zero, one, or more of the following clauses: `default`, `instance`, or `cell`.

`endconfig`

Is the keyword that ends a configuration.

The default Clause

The `default` clause specifies the logical libraries in which to search to resolve a default cell instance. A default cell instance is an instance in the design that is not specified in a subsequent `instance` or `cell` clause in the configuration.

You specify these libraries with the `liblist` keyword. The following is an example of a `default` clause:

```
default liblist lib1 lib2;
```

This `default` clause specifies resolving default instances in the logical libraries names `lib1` and `lib 2`.

Note:

- Do not enter a comma (,) between logical libraries.
- The default logical library work, if not listed in the list of logical libraries, is appended to the list of logical libraries and VCS searches the source files in work last.

The instance Clause

The `instance` clause specifies something about a specific instance. What it specifies depends on the use of the `liblist` or `use` keywords:

`liblist`

Specifies the logical libraries to search to resolve the instance.

`use`

Specifies that the instance is an instance of the specified cell in the specified logical library.

The following are examples of `instance` clauses:

```
instance top.dev1 liblist lib1 lib2;
```

This `instance` clause tells VCS to resolve instance `top.dev1` with the cells assigned to logical libraries `lib1` and `lib2`:

```
instance top.dev1.gm1 use lib2.gizmult;
```

This `instance` clause tells VCS that `top.dev1.gm1` is an instance of the cell named `gizmult` in logical library `lib2`.

The cell Clause

A cell clause is similar to an `instance` clause except that it specifies something about all instances of a cell definition instead of specifying something about a particular instance. What it specifies depends on the use of the `liblist` or `use` keywords:

`liblist`

Specifies the logical libraries to search to resolve all instances of the cell.

`use`

The specified cell's definition is in the specified library.

Hierarchical Configurations

A design can have more than one configuration. You can, for example, define a configuration that specifies the source code you use in particular instances in a subhierarchy, then you can define a configuration for a higher level of the design.

Suppose, for example, a subhierarchy of a design was an eight-bit adder and you have RTL Verilog code describing the adder in a logical library named `rtl1lib` and you have gate-level code describing the adder in a logical library named `gatel1ib`. If, for

example, you wanted the gate-level code used for the 0 (zero) bit of the adder and the RTL level code used for the other seven bits, the configuration might appear as:

```
config cfg1;
design aLib.eight_adder;
default liblist rtllib;
instance adder.fulladd0 liblist gatelib;
endconfig
```

Now, if you were going to instantiate this eight-bit adder eight times to make a 64-bit adder, you would use configuration `cfg1` for the first instance of the eight-bit adder, but not in any other instance. A configuration that would perform this function is as follows:

```
config cfg2;
design bLib.64_adder;
default liblist bLib;
instance top.64add0 use work.cfg1:config;
endconfig
```

The -top Compile-time Option

VCS has the `-top` compile-time option for specifying the configuration that describes the top-level configuration or module of the design, for example:

```
vcs -top top_cfg +v2k ...
vcs -top test -sverilog ...
```

The `-top` compile-time option requires the `+v2k` or `-sverilog` compile-time option.

If you have coded your design to have more than one top-level module, you can enter more than one `-top` option, or you can append arguments to the option using the plus delimiter. For example:

```
-top top_cfg+test+
```

Using the `-top` options tells VCS not to create extraneous top-level modules, that is, one that you don't specify.

Limitations of Configurations

In the current implementation, V2K configurations have the following limitations:

- You cannot specify source code for user-defined primitives in a configuration.
- The VPI functionality, described in section 13.6 "Displaying library binding information" in the Std 1364-2001 IEEE Verilog Hardware Description LRM, is not implemented.
- Array of instances is not supported.

Using `+evalorder` Option

VCS uses the `+evalorder` option to evaluate the active events when limiting the exposure of race conditions present in the design.

VCS divides the active events in the following categories:

- Combinational events: evaluates combinational logic such as gates, continuous assigns, and combinational UDPs.

- Behavioral events: evaluates behavioral logic such as always blocks, initial blocks, tasks, etc.

VCS first evaluates all the events in the combinational queue, and evaluates the events in the behavioral queue. If the behavioral events trigger more combinational events, VCS evaluates them only after the events in the behavioral queue are evaluated. This masks the race conditions happening at the boundaries of the combinational and behavioral parts of the design.

In this example, VCS without the `+evalorder` option will process the continuous assign statement after the statement `q = 0` or add it to the active events queue for later processing. Therefore, `$display` will show either 0 or X as the value of p.

```
module eval();
  wire p;
  reg q;
  assign p = q;
  initial
    begin
      #1 q = 0;
      $display("Value of p is %b", p);
    end
  endmodule
```

With the `+evalorder` option, VCS changes the scheduling of the continuous assignment to happen after all events in the initial block are done. Therefore, `$display` will always display the previous value of p, which is X.

Compiling the Design

4-20

5

Simulating the Design

This chapter describes the following:

- “Using DVE”
- “Using UCLI”
- “Key Runtime Features”

As described in the section “[Simulation](#)” on page 2-10, you can simulate your design in either interactive or batch mode. To simulate your design in interactive mode, you need to use DVE or UCLI. To simulate your design in batch mode, refer to the section entitled, “[Batch Mode](#)” on page 2-11.

Using DVE

DVE provides you with a graphical user interface to debug your design. Using DVE, you can debug the design in interactive mode or in post-processing mode. In the interactive mode, apart from running the simulation, DVE allows you to do the following:

- View waveforms
- Trace Drivers and loads
- Schematic and Path Schematic view
- Compare waveforms
- Execute UCLI/Tcl commands
- Set line, time, or event breakpoints
- Line stepping

However, in post-processing mode, a VPD/VCD/EVCD file is created during simulation, and you use DVE to:

- View waveforms
- Trace Drivers and loads
- Schematic and Path Schematic view
- Compare waveforms

Use the following command to invoke the simulation in interactive mode using DVE:

```
% simv -gui
```

Use the following command to invoke DVE in post-processing mode:

```
% dve -vpd [VPD/EVCD_filename]
```

For information on generating a VPD/EVCD dump file, see “[VPD, VCD, and EVCD Utilities](#)” on page 7-1.

For more information on using DVE, see “[Using Discovery Visual Environment](#)” on page 8-1.

Using UCLI

Unified Command-line Interface (UCLI) provides a common set of commands for interactive simulation. Using UCLI commands, you can do the following:

- Control the simulation
- Dump a VPD file
- Save/Restore the simulation state
- Force/Release a signal
- Debug the design using breakpoints, scope/thread information, built-in macros

UCLI commands are built based on Tcl. Therefore, you can execute any Tcl command or procedures at the UCLI prompt. This provides you with more flexibility to debug the design in interactive mode. See “[Using Unified Command-line Interface](#)” on page 9-1 for a list of commonly used UCLI commands. You can also refer to the *UCLI User Guide* for a complete list and descriptions of the UCLI commands.

The following command starts the simulation from the UCLI prompt:

```
% simv -ucli
```

When you execute the above command, VCS takes you to the UCLI command prompt, `ucli>`, as shown below:

```
% simv -ucli  
ucli%
```

At this prompt, you can execute any UCLI command to debug or run the simulation. You also can specify the list of required UCLI commands in a file, and source it to the UCLI prompt or specify the file as an argument to the runtime option, `-do`, as shown below:

```
% simv -ucli  
ucli% source file.cmds  
  
% simv -ucli -do file.cmds
```

The following section describes some of the commonly used UCLI commands and provides examples. For a complete list, see [Chapter 9, "Using Unified Command-line Interface"](#).

Key Runtime Features

Key runtime features includes:

- [“Passing Values from the Runtime Command Line”](#)
- [“Profiling the Simulation”](#)
- [“Save and Restart The Simulation”](#)
- [“Specifying a Long Time Before Stopping The Simulation”](#)

- “How VCS Prevents Time 0 Race Conditions”

Passing Values from the Runtime Command Line

The `$value$plusargs` system function can pass a value to a signal from the `simv` runtime command line using a `plusarg`. The syntax is as follows:

```
integer = $value$plusargs("plusarg_format",signalname);
```

The `plusarg_format` argument specifies a user-defined runtime option for passing a value to the specified signal. It specifies the text of the option and the radix of the value that you pass to the signal. The following code example contains this system function:

```
module valueplusargs;
reg [31:0] r1;
integer status;

initial
begin
$monitor("r1=%0d at %0t",r1,$time);
#1 r1=0;
#1 status=$value$plusargs("r1=%d",r1);
end
endmodule
```

If you enter the following `simv` command line:

```
% simv +r1=10
```

The `$monitor` system task displays the following:

```
r1=x at 0  
r1=0 at 1  
r1=10 at 2
```

Profiling the Simulation

If you include the `+prof` compile-time option when you compile your design, VCS generates the `vcs.prof` file during simulation. This file contains a profile of the simulation in terms of the CPU time and memory that it uses.

For CPU time it reports the following:

- The percentage of CPU time used by the VCS kernel, the design, the SystemVerilog testbench program block, cosimulation applications using either the DPI or PLI, and the time spent writing a VCD or VPD file.
- The module instances in the hierarchy that use the most CPU time
- The module definitions whose instances use the most CPU time
- The Verilog constructs in those instances that use the most CPU time

For memory usage it reports the following:

- The amount of memory and the percentage of memory used by the VCS kernel, the design, the SystemVerilog testbench program block, cosimulation applications using either the DPI or PLI, and the time spent writing a VCD or VPD file.
- The amount of memory and the percentage of memory that each module definition uses.

You can use this information to see where in your design you might be able to modify your code for faster simulation performance.

The profile data in the `vcs.prof` file is organized into a number of “views” of the simulation. The `vcs.prof` file starts with views on CPU time, followed by views on memory usage.

CPU Time Views

The views on CPU time are as follows:

- “The Top-level View”
- “The Module View”
- “The Program View”
- “The Instance View”
- “The Program to Construct Mapping View”
- “The Top-level Construct View”
- “The Construct View Across Design”

The Top-level View

This view displays how much CPU time was used by:

- Any PLI application that executes along with VCS.
- VCS for writing VCD and VPD files.
- VCS for internal operations that can't be attributed to any part of your design.
- The Verilog modules in your design.

- A SystemVerilog testbench program block, if used.

Example 5-1 Top-level View

TOP-LEVEL VIEW	
TYPE	%Totaltime
DPI	0.00
PLI	0.00
VCD	0.00
KERNEL	29.06
MODULES	51.87
PROGRAMS	21.17
PROGRAM GC	1.64

In this example, there is no PLI application and VCS does not write a VCD or VPD file. VCS used 51.87% of the CPU time to simulate the design, 21.94% for a testbench program, and 29.06% for internal operations, such as scheduling, that VCS cannot attribute to any part of the design. The designation KERNEL is for these internal operations. PROGRAM GC is for the garbage collector.

The designation VCD is for the simulation time used by the callback mechanisms inside VCS for writing either VCD or VPD files.

If there was CPU time used by a PLI application, you could use a tool such as gprof or Quantify to profile the PLI application.

The Module View

This view displays the module definitions whose instances use the most CPU time. It does not list module definitions whose module instances collectively use less than 0.5% of the CPU time.

Example 5-2 Module View

```
=====
          MODULE VIEW
=====
Module(index)           %Totaltime   No of Instances  Definition
-----
FD2                   (1)        62.17        10000    /u/design/design.v:142.
EN                    (2)        8.73         1000     /u/design/design.v:131.
-----
```

In this example, there are two module definitions whose instances collectively used a significant amount of CPU time, modules FD2 and EN.

The profile data for module FD2 is as follows:

- FD2 has an index number of 1. Other views that show the hierarchical names of module instances use this index number. The index number associates a module instance with a module definition because module identifiers do not necessarily resemble the hierarchical names of their instances.
- The instances of module FD2 used 62.17% of the CPU time.
- There are 10,000 instances of module FD2. The number of instances is a way to assess the CPU time used by these instances. For example, as in this case, a high CPU time with a correspondingly high number of instances tells you that each instance isn't using very much CPU time.
- The module header, the first line of the module definition, is in source file design.v on line 142.

The Program View

The program view displays the simulation time used by the testbench program, the number of instances, and the line number where it starts in its source file.

Example 5-3 Program View

PROGRAM VIEW				
Program(index)	%Totaltime	No of Instances	Definition	
test	(1) 21.17	1	/u/design/test.sv:25.	

The Module to Construct Mapping View

This view displays the CPU time used by different types of Verilog constructs in each module definition in the module view. The following lists the different types of Verilog constructs:

- always constructs (commonly called always blocks)
- initial constructs (commonly called initial blocks)
- module path delays in specify blocks
- timing check system tasks in specify blocks
- combinational logic including gates or built-in primitives and continuous assignment statements
- user-defined tasks
- user-defined functions
- module instance ports
- user-defined primitives (UDPs)
- Any Verilog code protected by encryption

Ports use simulation time particularly when there are expressions in port connection lists such as bit or part selects and concatenation operators.

This view has separate sections for the Verilog constructs for each module definition in the module view.

Example 5-4 Module to Construct Mapping View

```
=====
      MODULE TO CONSTRUCT MAPPING
=====
```

1. FD2

Construct type	%Totaltime	%Moduletime	LineNo
Always	27.44	44.14	design.v : 150-160.
Module Path	23.17	37.26	design.v : 165-166.
Timing Check	11.56	18.60	design.v : 167-168.

2. EN

Construct type	%Totaltime	%Moduletime	LineNo
Combinational	8.73	100.00	design.v: 137.

For each construct the view reports the percentage of “Totaltime” and “Moduletime”.

%Totaltime

The percentage of the total CPU time that was used by this construct.

%Moduletime

Each module in the design uses a certain amount of CPU time. This percentage is the fraction of the module’s CPU time that was used by the construct.

In the section for module FD2:

- An always block in this module definition used 27.44% of the TOTAL CPU time. Of all the CPU time consumed by all instances of the FD2 module, 44.14% is spent on this construct ($44.14\% \text{ of } 62.17\% = 27.44\%$). The always block is in source file `design.v` between lines 150 and 160.

If there were another always block in module FD2 that used more than 0.5% of the CPU time, there would be another line in this section for it, beginning with the `always` keyword.

- The module path delays in this module used 23.17% of the TOTAL CPU time. Of all the CPU time consumed by all instances of the FD2 module, 37.26% is spent on this construct. These module path delays can be found on lines 165-166 of the `design.v` source file.
- The timing check system tasks in this module used 11.56% of the TOTAL CPU time. Of all the CPU time consumed by all instances of the FD2 module, 18.60% is spent on this construct. These timing check system tasks can be found on lines 167-167 of the `design.v` source file.

In the section for module EN, a construct classified as Combinational used 8.73 of the total CPU time. 100% of the CPU time used by all instances of EN were used for this combinational construct.

No initial blocks, user-defined functions, or user-defined tasks, ports, UDPs, or encrypted code in the design used more than 0.5% of the CPU time. If there were, there would be a separate line for each of these types of constructs.

The Instance View

This view displays the module instances that use the most CPU time. An instance must use more than 0.5% of the CPU time to be entered in this view.

Example 5-5 Instance View

```
=====
           INSTANCE VIEW
=====
Instance          %Totaltime
-----
test.lfsr1000_1.lfsr100_1.lfsr10_1.lfsr_1.en_
1             ( 2 )        0.73
-----
```

In this example, there is only one instance that uses more than 0.5% of the CPU time.

This instance's hierarchical name is

test.lfsr1000_1.lfsr100_1.lfsr10_1.lfsr_1.en_1.
(Long hierarchical names wrap to the next line.)

The instance's index number is 2, indicating an instance of module EN, which had an index of 2 in the module view.

This instance used 0.73% of the CPU time.

No instance of module FD2 is listed here, so no individual instance of FD2 used more than 0.5% of the CPU time.

Note:

It is very common for no instances to appear in the instance view. This happens when many instances use some of the simulation time, but none use more than 0.5% of the total simulation time.

The Program to Construct Mapping View

The program to construct mapping view lists the testbench constructs that use the most simulation time and list the percentage of the total simulation they use, and the percentage of the program's simulation time each type of construct uses. It also lists the source file and line number of the constructs declaration.

Example 5-6 Program to Construct Mapping View

```
=====
PROGRAM TO CONSTRUCT MAPPING
=====

----- 1. test -----
Construct    Construct type    %Totaltime   %Programtime   LineNo
-----
name1::name2  Program Task      2.85        13.45       /u/design/vmm.sv : 12668-12901.
var          queue.var         2.64        12.45       /u/design/vmm.sv : 14551-14558.
name3::name4  Program Function  0.99        4.67       /u/design/vmm.sv : 13890-14215.
```

The Top-level Construct View

This view shows you the CPU time used by different types of constructs throughout the design.

Example 5-7 Top-level Construct View

TOP-LEVEL CONSTRUCT VIEW	
Construct	%Totaltime
Combinational	28.14
Task	16.58
Program Task	9.87
Always	6.52
Program Function	5.82
queue.size	2.64
Port	2.01
Object new	1.92
Initial	0.89
Program Thread	0.79
Function	0.76
queue.name	0.09
queue.name	0.05

The Construct View Across Design

This view displays the module or program definitions that contain a type of construct that used more than 0.5% of the CPU time. There are separate sections for each type of construct and each section contains a list of the modules or programs that contain that type of construct.

Example 5-8 Top-level Construct View

=====
CONSTRUCT VIEW ACROSS DESIGN
=====

1. Always

Module	%TotalTime
FD2	27.44

2. Module Path

Module	%TotalTime
FD2	23.17

3. Timing Check

Module	%TotalTime
FD2	11.56

4. Combinational

Module	%TotalTime
EN	8.73

Memory Usage Views

The views on memory usage are as follows:

- Top-level View
- Module View
- “The Program View”

The Top-level View

This view displays how much memory was used by:

- Any PLI or DPI application that executes along with VCS
- VCS for writing VCD and VPD files
- VCS for internal operations (known as the kernel) that can't be attributed to any part of your design
- The Verilog modules in your design
- A SystemVerilog testbench program block, if used

Example 5-9 Top-level View

```
=====
//      Simulation memory:    2054242 bytes
=====

=====
          TOP-LEVEL VIEW
=====
      TYPE        Memory   %Totalmemory
-----
          DPI         0       0.00
          PLI         0       0.00
          VCD         0       0.00
          KERNEL     890408   43.34
          MODULES    1163834  56.66
          PROGRAMS    0       0.00
-----//
```

Just before the top-level view, VCS writes the total amount of memory used by the simulation. In this example, it is 2054242 bytes.

In this example, there is no DPI or PLI application and VCS does not write a VCD or VPD file.

VCS used 1163834 bytes of memory and 56.66% of the total memory to simulate the design.

VCS used 890408 bytes of memory and 43.34% of the total memory for internal operations, such as scheduling, that can't be attributed to any part of the design. The designation KERNEL is for these internal operations.

The designation VCD is for the simulation time used by the callback mechanisms inside VCS for writing either VCD or VPD files.

The Module View

The module view shows the amount of memory used, and the percentage of memory used, by each module definition.

Example 5-10 Top-level View

=====					
MODULE VIEW					
Module(index)		Memory	%Totalmemory	No of Instances	Definition
bigmem	(1)	1048704	51.05	2	expl.v:16.
bigtime	(2)	115030	5.60	2	expl.v:61.
test	(3)	100	0.00	1	expl.v:1.

=====

In this example, the instances of module `bigmem` used 1048704 bytes of memory, 51.05% of the total memory used. The instances of module `bigtime` used 115030 bytes of memory, 5.6% of the total memory used.

The Program View

The program view displays the amount of memory used, and the percentage of memory used, by each testbench program.

Example 5-11 Program View

PROGRAM VIEW				
Program(index)	Memory	%Totalmemory	No of Instances	Definition
test	(1) 4459091	18.74	1	/u/design/test.sv:25.

Save and Restart The Simulation

VCS provides a save and restart feature using `$save` and `$restart` system tasks. These system tasks allows you to save the checkpoints of the simulation at arbitrary times. The resulting checkpoint files can be executed at a later time, causing simulation to resume at the point immediately following the save.

Note:

Save and Restart using `$save` and `$restart` system task is for designs having both DUT and the testbench in Verilog HDL. You can also use the UCLI `save` and `restart` feature. For more information, see the *Unified Command-line Interface User Guide*, available in the online HTML documentation system.

Benefits of save and restart include:

- Regular checkpoints for interactively debugging problems found during long batch runs
- Use of plusargs to start action such as `$dumpvars` on restart

- Execution of common simulation system tasks such as \$reset just once in a regression

Restrictions of save and restart include:

- Requires extra Verilog code to manage the save and restart
- Must duplicate start-up code if handling plusargs on restart
- File I/O suspend and resume in PLI applications must be given special consideration

Save and Restart Example

Example 5-12 illustrates the basic functionality of save and restart.

The \$save call does not execute a save immediately, but schedules the checkpoint save at the end of the current simulation time just before events scheduled with #0 are processed. Therefore, events delayed with #0 are the first to be processed upon restart.

Example 5-12 Save and Restart Example

```
% cat test.v
module simple_restart;
initial begin
    #10
    $display("one");
    $save("test.chk");
    $display("two");
    #0 // make the following occur at restart
    $display("three");
    #10
    $display("four");
end
endmodule
```

Now compile the example source file:

```
% vcs test.v
```

Now run the simulation:

```
% simv
```

VCS displays the following:

```
one
two
$save: Creating test.chk from current state of simv...
three
four
```

To restart the simulation from the state saved in the check file, enter:

```
% test.chk
```

VCS displays the following:

```
Restart of a saved simulation
three
four
```

Save and Restart File I/O

VCS remembers the files you opened via `$fopen` and reopens them when you restart the simulation. If no file with the old file name exists, VCS opens a new file with the old file name. If a file exists having the same name and length at the time you saved the old file, then VCS appends further output to that file. Otherwise, VCS attempts to open a file with a file name equal to the old file name plus the suffix `.N`. If a file with this name already exists, VCS exits with an error.

If your simulation contains PLI routines that do file I/O, the routines must detect both the save and restart events, closing and reopening files as needed. You can detect save and restart calls using `misctf` callbacks with reasons `reason_save` and `reason_restart`.

When running the saved checkpoint file, be sure to rename it so that further `$save` calls do not overwrite the binary you are running. There is no way from within the Verilog source code to determine if you are in a previously saved and restarted simulation, therefore, you cannot suppress the `$save` calls in a restarted binary.

Save and Restart With Runtime Options

If your simulation behavior depends on the existence of runtime `plusargs` or any other runtime action (such as reading a vector file), be aware that the restarted simulation uses the values from the original run unless you add special code to process runtime events after the restart action. Depending on the complexity of your environment and your usage of the save and restart feature, this can be a significant task.

For example, if you load a memory image with `$loadmemb` at the beginning of the simulation and want to be able to restart from a checkpoint with a different memory image, you must add Verilog code to load the memory image after every `$save` call. This ensures that at the beginning of any restart the correct memory image is loaded before simulation begins. A reasonable way to manage this is to create a task to handle processing arguments, and call this task at the start of execution, and after each save.

The following example illustrates this in greater detail. The first run optimizes simulation speed by omitting the +dump flag. If a bug is found, the latest checkpoint file is run with the +dump flag to enable signal dumping.

```
// file test.v
module dumpvars();
task processargs;
    begin
        if ($test$plusargs("dump")) begin
            $dumpvars;
        end
    end
end task
//normal start comes here
initial begin
    processargs;
end
// checkpoint every 1000 time units
always
    #1000 begin
        // save some old restarts
        $system("mv -f save.1 save.2");
        $system("mv -f save save.1");
        $save("save");
        #0 processargs;
    end
endmodule
// The design itself here
module top();
    ....
endmodule
```

Specifying a Long Time Before Stopping The Simulation

You can use the `+vcs+stop+time` runtime option to specify the simulation time when VCS halts simulation. This works if the *time* value you specify is less than 2^{32} or 4,294,967,296. You can also use the `+vcs+finish+time` runtime option to specify when VCS either halts or ends simulation, provided that the time value is less than 2^{32} .

For *time* values greater than 2^{32} , you must follow a special procedure that uses two arguments with the `+vcs+stop` or `+vcs+finish` runtime options. This procedure is as follows:

1. Subtract 2×2^{32} from the large *time* value.

For example, if you want a time value of 10,000,000,000 (10 billion):

$$10,000,000,000 - (2^4,294,967,296) = (1,410,065,408)$$

This difference is the first argument.

VCS can do some of this work for you by using the following source code:

```
module wide_time;
  time wide;
  initial
  begin
    wide = 64'd10_000_000_000;
    $display("Hi=%0d, Lo=%0d", wide[63:32], wide[31:0]);
  end
endmodule
```

VCS displays:

Hi=2, Lo=1410065408

2. Divide the large *time* value by 2^{32} .

In this example:

$$\frac{10,000,000,000}{4,294,967,296} = 2.33$$

3. Narrow down this quotient to the nearest whole number. This whole number is the second argument.

In this example, you would narrow down to 2.

You now have the first and second argument. Therefore, in this example, to specify stopping simulation at time 10,000,00,000 you would enter the following runtime option:

```
+vcs+stop+1410065408+2
```

How VCS Prevents Time 0 Race Conditions

At simulation time 0, VCS always executes the always blocks where any of the signals in the event control expression, that follows the `always` keyword (the sensitivity list), initializes at time 0.

For example, consider the following code:

```
module top;
reg rst;
wire w1,w2;
initial
rst=1;
bottom bottom1 (rst,w1,w2);
endmodule

module bottom (rst,q1,q2);
output q1,q2;
input rst;
reg rq1,rq2;

assign q1=rq1;
assign q2=rq2;

always @ rst
begin
rq1=1'b0;
rq2=1'b0;
$display("This always block executed!");
end
endmodule
```

With other Verilog simulators there are two possibilities at time 0:

- The simulator executes the initial block first, initializing `reg rst`, then the simulator evaluates the event control sensitivity list for the `always` block and executes the `always` block because the simulator initialized `rst`.

The simulator evaluates the event control sensitivity list for the `always` block, and so far, `reg rst` has not changed value during this time step, therefore, the simulator does not execute the `always` block. Then the simulator executes the `initial` block and initializes `rst`. When this occurs, the simulator does not re-evaluate the event control sensitivity list for the `always` block.

Simulating the Design

5-28

6

VCS Multicore Technology Application Level Parallelism

VCS Multicore Technology takes advantage of the computing power of multiple processors in one machine to improve simulation turnaround time.

Use the following VCS Multicore Technology options in a simulation:

- Assertion profiling and simulation
- Toggle coverage
- Multicore functional coverage
- VPD dumping

VCS Multicore Technology Options

You use the VCS `-parallel` option to invoke parallel compilation. The syntax is:

```
vcs filename(s).v -parallel [ +multicore_option(s) ]  
[ -parallel+show_features ] [-o multicore_executable_name]  
[ vcs-options ]
```

These options and properties are as follows:

`-parallel`

When used without VCS Multicore options, `-parallel` enables all VCS Multicore Technology options. When used with VCS Multicore options, `-parallel` enables only those option specified.

This option is available at compile-time only.

`+fc [=NCONS]`

This compile-time option enables multicore Functional Coverage, and with *NCONS* specifies the number of PFC consumers. *NCONS* can be changed at run time. For example,

```
vcs -parallel+fc ...  
vcs -parallel+fc=3 ...
```

`+profile`

Enables application level profiling.

`+sva [=NCONS]`

This compile-time option enables multicore SVA, and with *NCONS* specifies the number of multicore SVA consumers. *NCONS* can be changed at run time.

`+tgc [=NCONS]`

Enables multicore Toggle Coverage , and specifies the number of multicore toggle coverage consumers. To enable the use of the same executable for both serial and parallel runs, use this option at runtime.

`NCONS` specifies the number of multicore SVA consumers. For ALP, `NCONS` can be changed at run time.

`+vpd [=NCONS]`

Enables multicore VCD+ Dumping. `NCONS` specifies the number of multicore SVA consumers. For ALP, `NCONS` can be changed at run time

`+vpd_sidebuf=MULT`

Sets the size of PVPD side buffer to `MULT` times the size of the main buffer. To enable the use of the same executable for both serial and parallel runs, use this option at runtime.

`[-o multicore_executable_name]`

Using the VCS `-o` option to specify the simulation executable binary filename allows work on multiple simultaneous VCS Multicore compiles and runs. VCS Multicore-specific data is stored in a directory `executable_name.pdaidir`. The default path name is `simv.pdaidir`.

Note:

If [`NCONS`] is not specified, the default is 1 client. For ALP, `NCONS` can be changed at run time.

`-parallel+show_features`

Displays enabled VCS Multicore features. Note that you must enter the `-parallel` option with `+show_features`

Examples:

```
-parallel+vpd is equal to -parallel+vpd=1  
-parallel+tgl is equal to -parallel+tgl=1
```

VCS Multicore option examples:

```
vcs -parallel+fc .... -o psimv  
vcs -parallel+vpd+fc -parallel+tgl -o par_simv ....  
vcs -parallel+design=part.cfg+sva ....
```

Use Model for Assertion Simulation

1. Run VCS Multicore compilation specifying the `sva` option.
2. Run VCS Multicore simulation.

Use Model for Toggle and Functional Coverage

1. Run VCS Multicore compilation specifying the VCS Multicore `tgl` option and coverage metric options for toggle coverage, and/or the VCS Multicore `fc` option for functional coverage. You can optionally specify the number of consumers for each.
2. Run the simulation to generate coverage results.
3. Generate coverage result reports.

Use Model for VPD Dumping

1. Run VCS Multicore compilation specifying the `vpd` option.
2. Run the simulation to generate the VPD file.

Running VCS Multicore Simulation

VCS Multicore Technology takes advantage of the computing power of multiple processors to improve simulation turnaround time

You can generate results for one of all the following VCS Multicore Technology options in a simulation:

- Assertion simulation
- Toggle coverage
- Functional coverage
- VPD file generation

Assertion Simulation

You can process only assertion level results or assertion level results along with other VCS Multicore options.

1. Compile using the VCS Multicore `-parallel` option, the assertion compilation option or options, and other VCS Multicore and VCS options.

```
vcs filename(s).v -parallel+ [sva [=NCONS] ]  
[-ntb_opts] [ multicore_options vcs_options ]
```

2. Run the simulation with VCS and VCS Multicore run-time options.

```
simv
```

Toggle Coverage

Generate results for only toggle coverage or toggle coverage along with other results by compiling the design with VCS Multicore options that include the `+tgl` option and VCS coverage metrics options. You can use the `+count` option to report total executed transactions. After generating coverage results, you can examine them using the Unified Report Generator.

Note:

To enable the use of the same executable for both serial and parallel runs, use this option at runtime.

`tgl [+count]`

Report total executed transactions.

1. Compile using the VCS Multicore `-parallel` option, coverage option or options, and other VCS Multicore and VCS options.

```
vcs filename(s).v -parallel+tgl[=NCONS] -cm tgl  
[multicore_options] [vcs_options]
```

2. Run the simulation to generate coverage results.

```
simv -cm tgl [vcs_options]
```

3. Generate coverage result reports:

```
urg -dir coverage_directory.cm urg_options
```

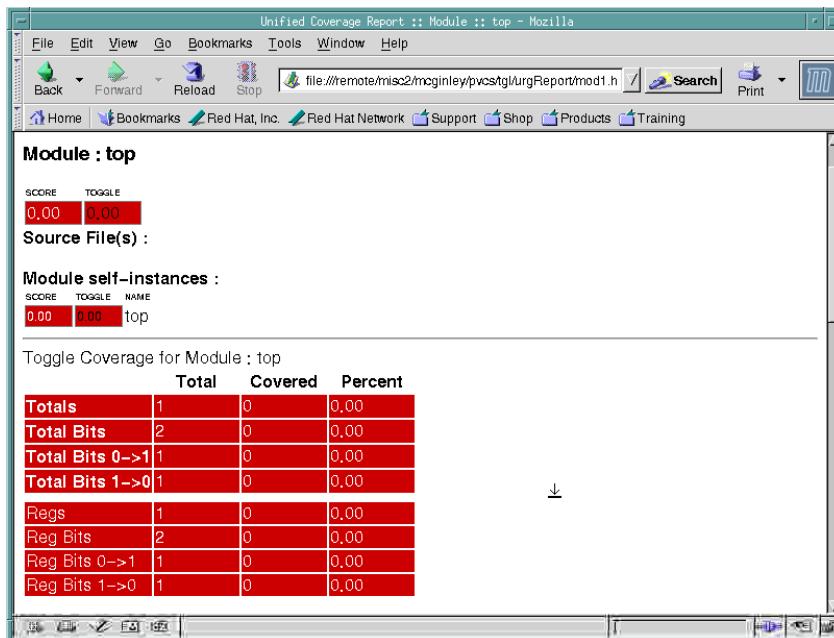
Example

In this example, toggle coverage results only are generated and the URG report is produced in the default HTML format.

```
% vcs -cm_tgl mda -q -cm_dir pragmaTest1.cm -cm tgl -sverilog  
-parallel+tgl=2 pragmaTest1.v  
% simv -cm tgl
```

```
% vcs -cm_pp -cm_dir pragmaTest1.cm
% urg -dir pragmaTest1.cm
```

Results can then be examined in your default browser.



Functional Coverage

Generate results for only functional coverage or functional coverage along with other results by compiling the design with VCS Multicore options that include the `+fc` option and VCS coverage metrics options. After generating coverage results, you can examine them using the Unified Report Generator.

1. Compile using the VCS Multicore `-parallel` option, coverage option or options, and other VCS Multicore and VCS options.

```
vcs filename(s).v -sverilog -parallel+fc [=NCONS]
[parallel_vcs_options] [vcs_options]
```

2. Run the simulation to generate coverage results.

simv

3. Generate coverage result reports:

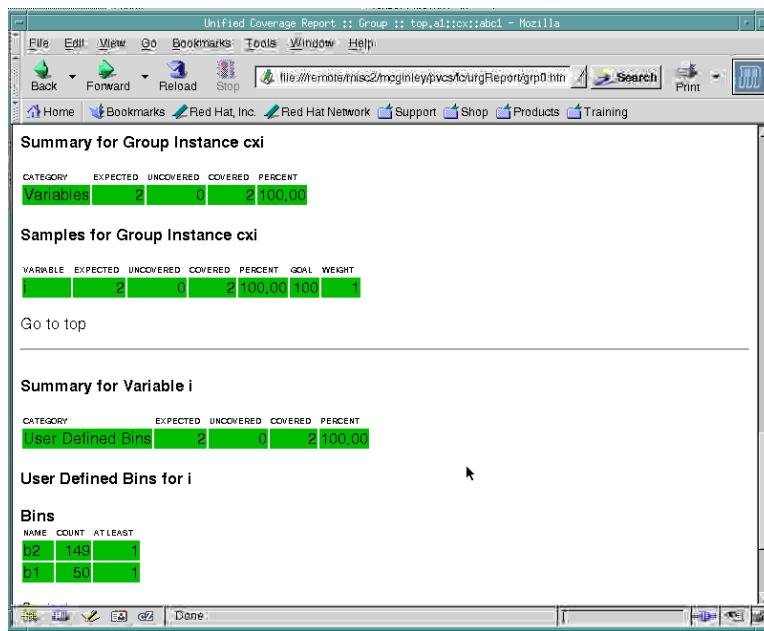
```
urg -dir coverage_directory.cm urg_options
```

Example

In this example, functional coverage results only are generated and the URG report is produced in the default HTML format.

```
% vcs iemIntf.v -ntb_opts dtm -sverilog -parallel+fc=2
% simv -covg_cont_on_error
% $urg -dir simv.vdb
% cat urgReport/gr*
%
```

Results can then be examined in your default browser.



VPD File

You can enable VCS Multicore VPD+ Dumping and specify the number of VCS Multicore VPD+ consumers using the VCS Multicore `vpd` option. To enable the use of the same executable for both serial and parallel runs, use this option at runtime.

Note:

When used with multiple consumers, VPD file size blow up might be an issue. Use `-parallel+vpd_buffer=<N>`, where N=256, 512 etc.

1. Compile using the VCS Multicore `-parallel` option with the `vpd [=NCONS]` option, and other VCS Multicore and VCS options.

```
vcs filename(s).v -debug_pp -parallel+vpd [=NCONS]  
[multicore_options] [vcs_options]
```

2. Run the simulation.

```
simv
```

You can post-process the results with the generated +VPD database.

Example

In this example, a VPD+ file with three specified consumers is generated.

```
% vcs -debug_pp -parallel+vpd=3 design.v  
% simv
```


7

VPD, VCD, and EVCD Utilities

This chapter describes the following:

- “Advantages of VPD”
- “Dumping a VPD File”
- “Dump Multi-dimensional Arrays and Memories”
- “Dumping an EVCD File”
- “Post-processing Utilities”

VCS allows you to save your simulation history in the following formats:

- Value Change Dumping (VCD)

VCD is the IEEE Standard for Verilog designs. You can save your simulation history in VCD format by using the `$dumpvars` Verilog system task.

- VCDPlus Dumping (VPD)

VPD is a Synopsys proprietary dumping technology. VPD has many advantages over the standard VCD ASCII format. See “[Advantages of VPD](#)” for more information. To dump a VPD file, use a system task, such as, UCLI and DVE. See “[Dumping a VPD File](#)” for more information.

- Extended VCD (EVCD)

EVCD dumps only the port information of your design. See “[Dumping an EVCD File](#)” for more information.

VCS also provides several post-processing utilities to:

- Convert VPD to VCD
- Convert VCD to VPD
- Merge VPD Files

Advantages of VPD

VPD offers the following significant advantages over the standard VCD ASCII format:

- Provides a compressed binary format that dramatically reduces the file size as compared to VCD and other proprietary file formats.
- The VPD compressed binary format dramatically reduces the signal load time.
- Allows data collection for signals or scopes to be turned on and off during a simulation run, therefore, dramatically improving simulation runtime and file size.

- Can save source statement execution data. This allows instant replay of source execution in the DVE Source Window.

To optimize VCS performance and VPD file size, consider the size of the design, the RAM memory capacity of your workstation, swap space, disk storage limits, and the methodology used in the project.

Dumping a VPD File

You can save your simulation history in VPD format in the following ways:

- [“Using System Tasks”](#) - For Verilog designs.
 - Using UCLI - For VHDL, Verilog, and mixed designs. See [“Dumping a VPD File” on page 9-7](#).
 - Using DVE - See the *Discovery Visual Environment User Guide*.
-

Using System Tasks

VCS provides Verilog system tasks to:

- [“Enable and Disable Dumping”](#)
- [“Override the VPD Filename”](#)
- [“Dump Multi-dimensional Arrays and Memories”](#)
- [“Capture Delta Cycle Information”](#)

Enable and Disable Dumping

You can use the Verilog system task `$vcdpluson` and `$vcdplusoff` to enable and disable dumping the simulation history in VPD format.

Note:

The default VPD filename is `vcdplus.vpd`. However, you can use `$vcdplusfile` to override the default filename, see [“Override the VPD Filename”](#).

\$vcdpluson

The following displays the syntax for `$vcdpluson`:

```
$vcdpluson (level/"LVL=integer", scope*, signal*) ;
```

Usage:

level | LVL=integer_variable

Specifies the number of hierarchy scope levels to descend to record signal value changes (a zero value records all scope instances to the end of the hierarchy; the default is zero).

You can also specify the number of hierarchy scope levels using "LVL=*integer_variable*". In this example, the *integer_variable* specifies the level to descend to record signal value changes.

scope

Specifies the name of the scope in which to record signal value changes (the default is all).

signal

Specifies the name of the signal in which to record signal value changes (the default is all).

Note:

In the syntax, * indicates that the argument can have a list of more than one value (for scopes or signals).

Example 1: Record all signal value changes.

```
'timescale 1ns/1ns
module test ();
...
initial
$vcpluson;
...
endmodule
```

When you simulate the above example, VCS saves the simulation history of the whole design in `vcplus.vpd`. For information on the use model to simulate the design, see “[Basic Usage Model](#)” on page [1-8](#).

**Example 2: Record signal value changes for scope
test.risc1.alureg and all levels below it.**

```
'timescale 1ns/1ns
module test ();
...
risc1 risc(...);

initial
$vcplusplus(test.risc1.alureg);

...
endmodule
```

When you simulate the previous example, VCS saves the simulation history of the instance alureg, and all instances below alureg in vcdplus.vpd.

\$vcplusplusoff

The \$vcplusplusoff task stops recording the signal value changes for specified scopes or signals.

The following displays the syntax for vcdplusplusoff:

```
$vcplusplusoff (level | "LVL=integer", scope*, signal*) ;
```

Example 1: Turn recording off.

```
'timescale 1ns/1ns
module test ();
...
initial
begin
    $vcdpluson; // Enable Dumping
    #5 $vcdplusoff; //Disable Dumping after 5ns
    ...
end
...
endmodule
```

The above example, enables dumping at 0ns, and disables dumping after 5ns.

Example 2: Stop recording signal value changes for scope test.risc1.alu1.

```
'timescale 1ns/1ns
module test ();
...
initial
begin
    $vcdpluson; // Enable Dumping
    $vcdplusoff(test.risc1.alu1); //Does not dump signal value
                                  //changes in test.risc1.alu1
    ...
end
...
endmodule
```

The above example, enables dumping on the entire design. However, \$vcdplusoff disables dumping the instance alu1 and instances below alu1.

Note:

If multiple \$vcdpluson commands cause a given signal to be saved, the signal will continue to be saved until an equivalent number of \$vcdplusoff commands are applied to the signal.

Override the VPD Filename

By default, \$vcdpluson writes the simulation history in the vcdplus.vpd file. However, you can override the default filename by using the system task \$vcdplusfile.

The syntax is as shown below:

```
$vcdplusfile ("filename");
```

Example:

```
'timescale 1ns/1ns
module test ();
...
initial
begin
    $vcdpluson; // Enable Dumping
    $vcdplusfile("my.vpd"); //Dumps signal value changes
                           //in my.vpd
    ...
end
...
endmodule
```

The above example writes the signal value changes of the whole design in my.vpd.

Dump Multi-dimensional Arrays and Memories

This section describes system tasks and functions that provide visibility into multi-dimensional arrays (MDAs).

There are two ways to view MDA data:

- The first method, which uses the `$vcdplusmemon` and `$vcdplusmemoff` system tasks, records data each time an MDA has a data change.

Note:

You should use the compilation options `+memcbk` and `+v2k` to use these system tasks.

- The second method, which uses the `$vcdplusmemorydump` system task, stores data only when the task is called.

Syntax for Specifying MDAs

Use the following syntax to specify MDAs using the `$vcdplusmemon`, `$vcdplusmemoff`, and `$vcdplusmemorydump` system tasks:

```
system_task( Mda [, dim1Lsb [, dim1Rsb [, dim2Lsb [, dim2Rsb [, ... dimNLsb [, dimNRsb]]]]] ] );
```

Usage:

```
system_task
```

Name of the system task (required). It can be `$vcdplusmemon`, `$vcdplusmemoff`, or `$vcdplusmemorydump`.

Mda

Name of the MDA to be recorded. It must not be a part select. If there are no other arguments, then all elements of the MDA are recorded to the VPD file.

dim1Lsb

Name of the variable that contains the left bound of the first dimension. This is an optional argument. If there are no other arguments, then all elements under this single index of this dimension are recorded.

dim1Rsb

Name of the variable that contains the right bound of the first dimension. This is an optional argument.

Note:

The dim1Lsb and dim1Rsb arguments specify the range of the first dimension to be recorded. If there are no other arguments, then all elements under this range of addresses within the first dimension are recorded.

dim2Lsb

This is an optional argument with the same functionality as dim1Lsb, but refers to the second dimension.

dim2Rsb

This is an optional argument with the same functionality as dim1Rsb, but refers to the second dimension.

`dimNLsb`

This is an optional argument that specifies the left bound of the N th dimension.

`dimNRsb`

This is an optional argument that specifies the right bound of the N th dimension.

Note that MDA system tasks can take 0 or more arguments, with the following caveats:

- No arguments: The whole design is traversed and all memories and MDAs are recorded.

Note that this process may cause significant memory usage, and simulation drag.

- One argument: If the object is a scope instance, all memories/MDAs contained in that scope instance and its children will be recorded. If the object is a memory/MDA, that object will be recorded.

Examples

This section provides examples and graphical representations of various MDA and memory declarations using the `$vcdplusmemon` and `$vcdplusmemoff` tasks.

In this example, `mem01` is a three-dimensional array. It has $3 \times 3 \times 3$ (27) locations; each location is 8 bits in length, as shown in [Figure 7-1](#).

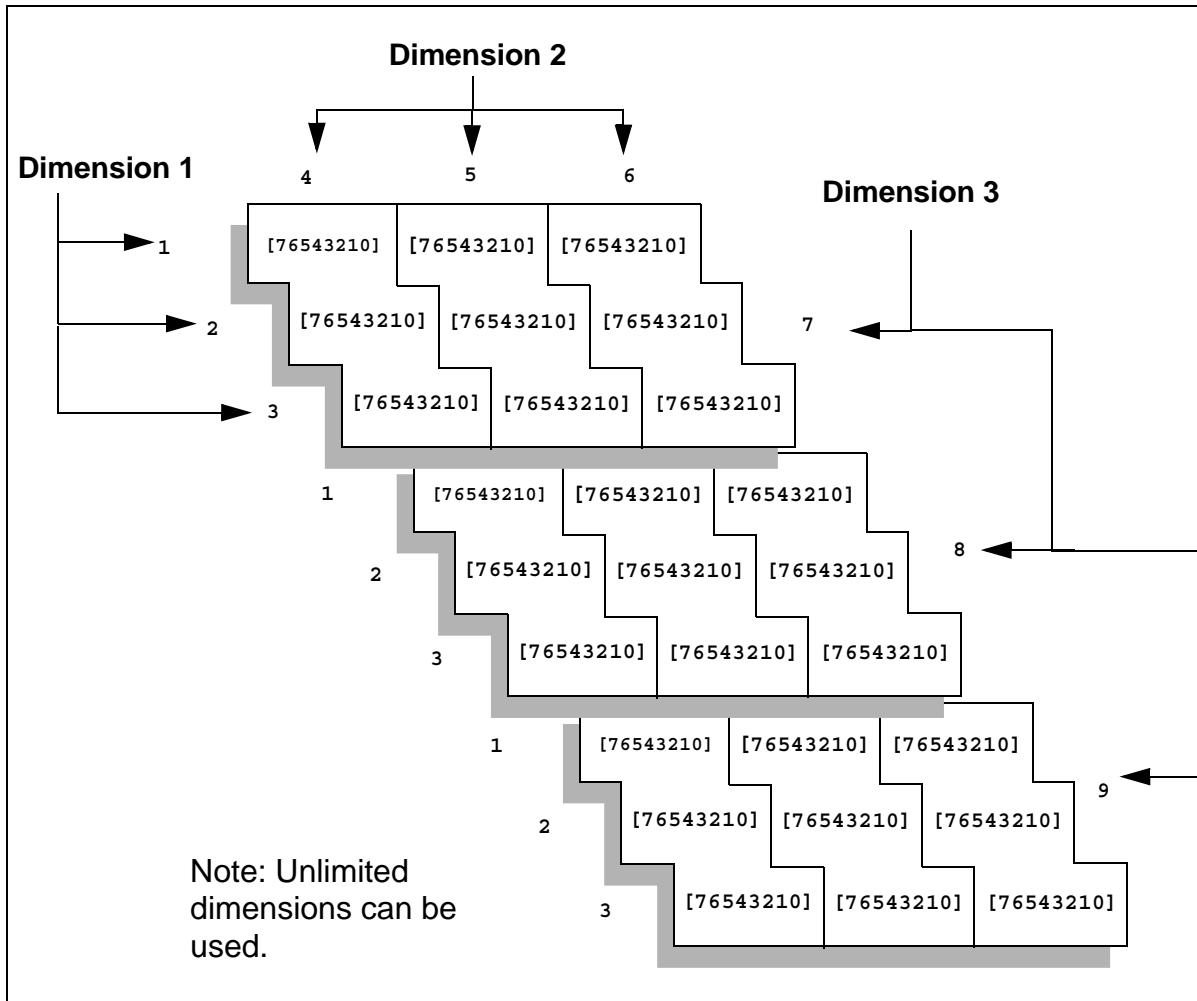
```
module tb();
...
reg [3:0] addr1L, addr1R, addr2L, addr2R, addr3L, addr3R;
reg [7:0] mem01 [1:3] [4:6] [7:9]
...
endmodule
```

Example 1: To dump all elements to the VPD File

```
module test();
...
initial
$vcplusmemon( mem01 );
    // Records all elements of mem01 to the VPD file.
...
endmodule
```

In the above example, `$vcplusmemon` dumps the entire `mem01` MDA.

Figure 7-1 reg [7:0] mem01 [1:3] [4:6] [7:9]

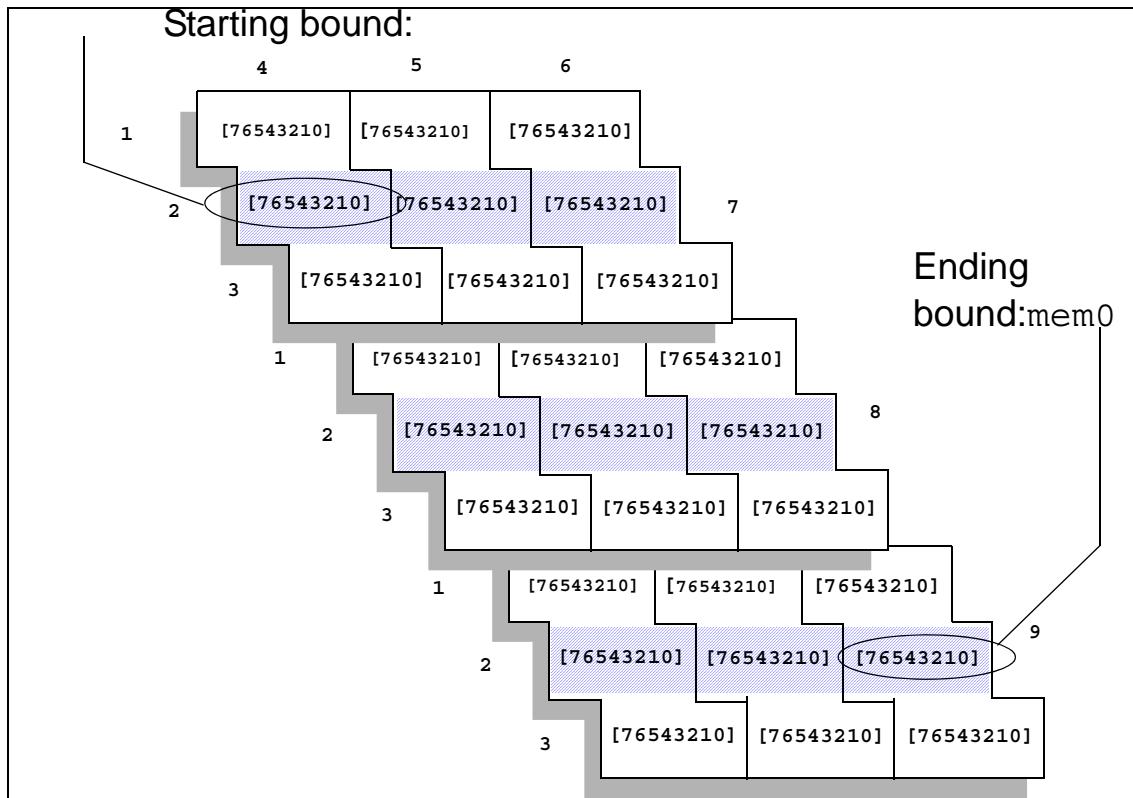


Example 2: To dump only mem01[2] to the VPD File

```
module test();
...
initial
begin
    addr1L = 2;
    $vcplusmemon( mem01, addr1L );
    // Records elements mem01[2][4][7] through mem01[2][6][9]
    ...
end
...
endmodule
```

The elements highlighted by the  in the following [Figure 7-2](#), illustrate this example.

Figure 7-2 `$vcdplusmemon(mem01, addr1L)`

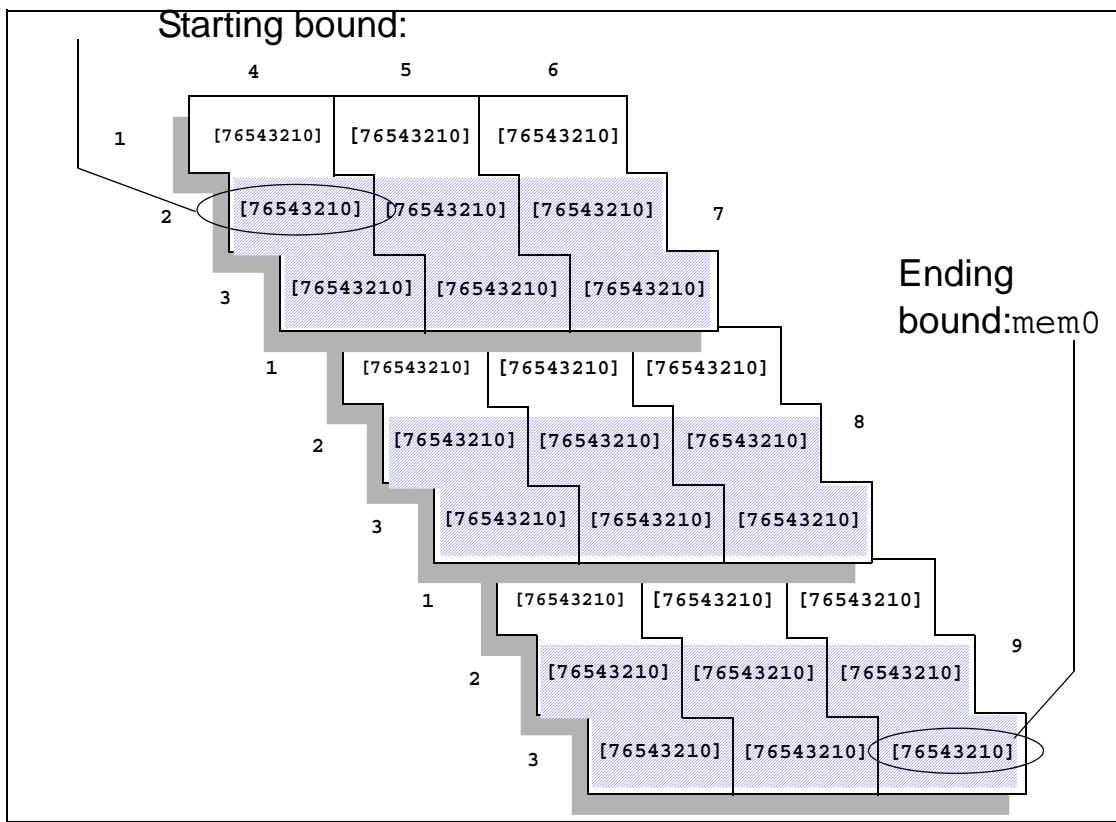


Example 3: To dump only `mem01[2][3]` to the VPD File

```
module test();
...
initial
begin
    addr1L = 2; addr1R = 3;
    $vcdplusmemon( mem01, addr1L, addr1R );
    // Records elements mem01[2] [4] [7] through mem01[3] [6] [9]
    ...
end
...
endmodule
```

The elements highlighted by the  in the following Figure 7-3, illustrate this example.

Figure 7-3 \$vcplusmemon(mem01, addr1L, addr1R)

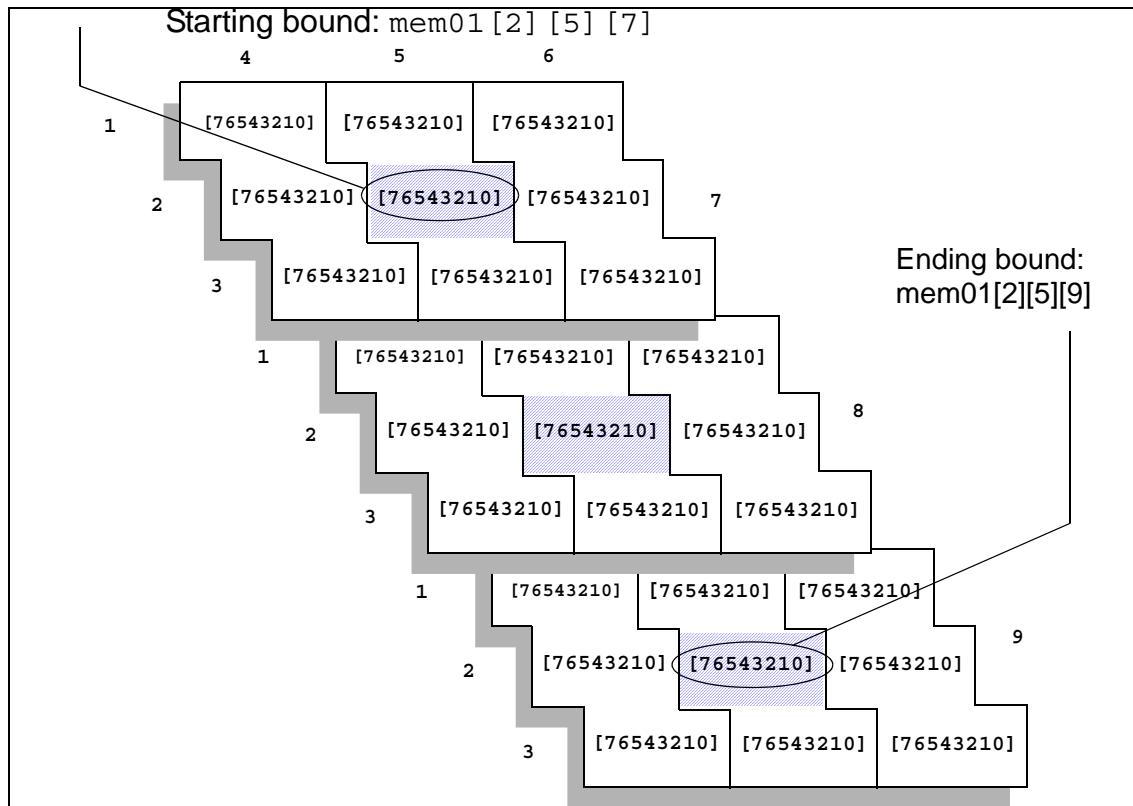


Example 4: To dump only mem01[2][2][5] to the VPD File

```
module test();
...
initial
begin
    addr1L = 2; addr1R = 2; addr2L = 5;
    $vcplusmemon( mem01, addr1L, addr1R, addr2L );
    // Records elements mem01[2] [5] [7] through mem01[2] [5] [9]
    ...
end
...
endmodule
```

The elements highlighted by the  in the following Figure 7-4, illustrate this example.

Figure 7-4 `$vcdplusmemon(mem01, addr1L, addr1R, addr2L)`



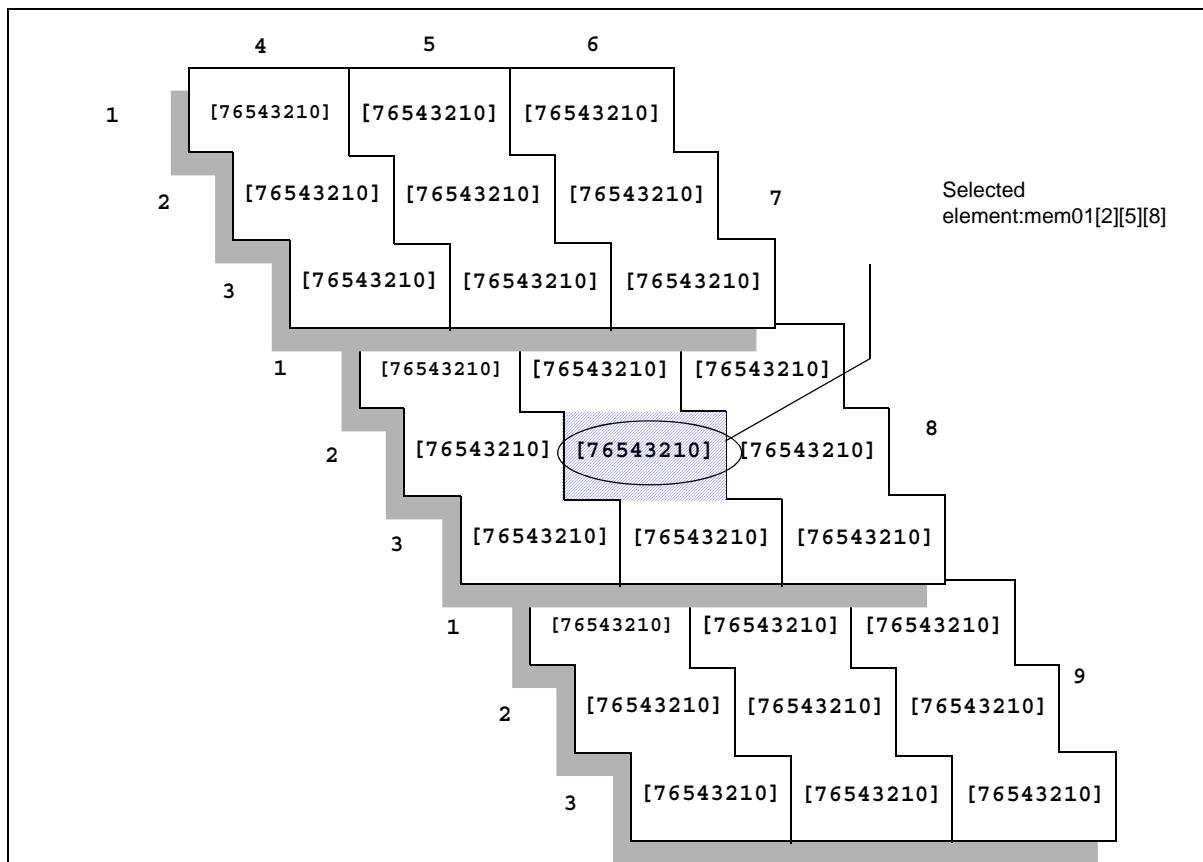
Example 5: To dump mem01[2][5][8] to the VPD File

```
module test();
...
initial
begin
    addr1L = 2;
    addr1R = 2;
    addr2L = 5;
    addr2R = 5;
    addr3L = 8;
    addr3R = 8;
    $vcdplusmemon( mem01, addr1L, addr1R, addr2L, addr2R,
```

```
addr3L, addr3R ) ;
    // Either command records element mem01[2][5][8]
    ...
end
...
endmodule
```

The elements highlighted by the  in the following Figure 7-5 illustrate this example.

Figure 7-5 \$vcplusmemon(mem01, addr1L, addr1R, addr2L, addr2R, addr3L, addr3R)



Using \$vcplusmemorydump

The `$vcplusmemorydump` task dumps a snapshot of memory locations. When the function is called, the current contents of the specified range of memory locations are recorded (dumped).

You can specify to dump the complete set of multi-dimensional array elements only once. You can specify multiple element subsets of an array using multiple `$vcplusmemorydump` commands, but they must occur in the same simulation time. In subsequent simulation times, `$vcplusmemorydump` commands must use the initial set of array elements or a subset of those elements. Dumping elements outside the initial specifications results in a warning message.

Capture Delta Cycle Information

You can use the following VPD system tasks to capture and display delta cycle information in the Waveform Window.

\$vcplusdeltacycleon

The `$vcplusdeltacycleon` task enables reporting of delta cycle information from the Verilog source code. It must be followed by the appropriate `$vcpluson`/`$vcplusoff` tasks.

Glitch detection is automatically turned on when VCS executes `$vcplusdeltacycleon` unless you have previously used `$vcplusglitchon/off`. Once you use `$vcplusglitchon/off`, DVE allows you explicit control of glitch detection.

Syntax:

```
$vcplusdeltacycleon;
```

Note:

Delta cycle collection can start only at the beginning of a time sample. The `$vcdplusdeltacycleon` task must precede the `$vcdplusion` command to ensure that delta cycle collection will start at the beginning of the time sample.

\$vcdplusdeltacycleoff

The `$vcdplusdeltacycleoff` task turns off reporting of delta cycle information starting at the next sample time.

Glitch detection is automatically turned off when VCS executes `$vcdplusdeltacycleoff` unless you have previously used `$vcdplusglitchon/off`. Once you use `$vcdplusglitchon/off`, DVE allows you explicit control of glitch detection.

Syntax:

```
$vcdplusdeltacycleoff;
```

Dumping an EVCD File

EVCD dumps the signal value changes of the ports at the specified module instance. You can dump an EVCD file, using the following system tasks:

```
$lsi_dumpports
```

For LSI certification of your design, this system task specifies recording a simulation history file that contains the transition times and values of the ports in a module instance.

This simulation history file for LSI certification contains more information than the VCD file specified by the `$dumpvars` system task. The information in this file includes strength levels and whether the test fixture module (test bench) or the Device Under Test (the specified module instance or DUT) is driving a signal's value.

Syntax:

```
$lsi_dumpports(module_instance, "filename");
```

Example:

```
$lsi_dumpports(top.middle1, "dumpports.dmp");
```

Instead, if you would prefer to have the `$lsi_dumpports` system task generate an extended VCD (EVCD) file, include the `+dumpports+ieee` runtime option.

```
$dumpports
```

Creates an EVCD file as specified in IEEE Standard 1364-2001 pages 339-340. You can, for example, input a EVCD file into TetraMAX for fault simulation. EVCD files are similar to the simulation history files generated by the `$lsi_dumpports` system task for LSI certification, but there are differences in the internal statements in the file. Further, the EVCD format is a proposed IEEE standard format whereas the format of the LSI certification file is specified by LSI.

Syntax:

```
$dumpports(module_instance, [module_instance,] "filename") ;
```

Example:

```
$dumpports(top.middle1, "dumpports.evcd") ;
```

If your source code contains a `$dumpports` system task and you want it to generate simulation history files for LSI certification, include the `+dumpports+lsi` runtime option.

Post-processing Utilities

VCS provides you with the following utilities to process VCD and VPD files. You can use these utilities to perform the following conversions:

- VPD file to a VCD file
- VCD file to a VPD file

- Merge a VPD file

Note:

All utilities are available in \$VCS_HOME/bin.

This section describes these utilities in the following sections:

- “The vcdpost Utility”
- “The vcdiff Utility”
- “The vcat Utility”
- “The vcsplit Utility”
- “[The vcd2vpd Utility](#)”
- “[The vpd2vcd Utility](#)”
- “[The vpdmerge Utility](#)”

The vcdpost Utility

You use the vcdpost utility to generate an alternative VCD file that has the following characteristics:

- Contains value change and transition times for each bit of a vector net or register, recorded as a separate signal. This is called “scalarizing” the vector signals in the VCD file.
- Avoids sharing the same VCD identifier code with more than one net or register. This is called “uniquifying” the identifier codes.

Scalarizing the Vector Signals

The VCD format does not support a mechanism to dump part of a vector. For this reason, if you enter a bit select or a part select for a net or register as an argument to the `$dumpvars` system task, VCS records value changes and transition times for the entire net or register in the VCD file. For example, if you enter the following in your source code:

```
$dumpvars(1,mid1.out1[0]);
```

In this example, `mid1.out1[0]` is a bit select of a signal because you need to examine the transition times and value changes of this bit. VCS however writes a VCD file that contains the following:

```
$var wire 8 ! out1 [7:0] $end
```

Therefore, all the value changes and simulation times for signal `out1` are for the entire signal and not just for the 0 bit.

The `vcddpost` utility can create an alternative VCD file that defines a separate `$var` section for each bit of the vector signal. The results are as follows:

```
$var wire 8 ! out1 [7] $end
$var wire 8 " out1 [6] $end
$var wire 8 # out1 [5] $end
$var wire 8 $ out1 [4] $end
$var wire 8 % out1 [3] $end
$var wire 8 & out1 [2] $end
$var wire 8 ' out1 [1] $end
$var wire 8 ( out1 [0] $end
```

What this means is that the new VCD file contains value changes and simulation times for each bit.

Uniquifying the Identifier Codes

In certain circumstances, to enable better performance, VCS assigns the same VCD file identifier code to more than one net or register, if these nets or registers have the same value throughout the simulation. For example:

```
$var wire 1 ! ramsel_0_0 $end
$var wire 1 ! ramsel_0_1 $end
$var wire 1 ! ramsel_1_0 $end
$var wire 1 ! ramsel_1_1 $end
```

In this example, VCS assigns the ! identifier code to more than one net.

Some back-end tools from other vendors fail when you input such a VCD file. You can use the `vcdpost` utility to create an alternative VCD file in which the identifier codes for all nets and registers, including those instances without value changes, are unique. For example:

```
$var wire 1 ! ramsel_0_0 $end
$var wire 1 " ramsel_0_1 $end
$var wire 1 # ramsel_1_0 $end
$var wire 1 $ ramsel_1_1 $end
```

The `vcdpost` Utility Syntax

The syntax for the `vcdpost` utility is as follows:

```
vcdpost [+scalar] [+unique] input_VCD_file output_VCD_file
```

Usage:

+scalar

Specifies creating separate \$var sections for each bit in a vector signal. This option is the default option and you include it on the command line when you also include the +unique option and want to create a VCD file that both scalarizes the vector nets and unquifies the identifier codes.

+unique

Specifies unquifying the identifier codes. When you include this option without the +scalar option, vcdpost unquifies the identifier codes without scalarizing the vector signals.

input_VCD_file

The name of the VCD file created by VCS.

output_VCD_file

The name of the alternative VCD file created by the vcdpost utility.

The vcdiff Utility

The vcdiff utility compares two dump files and reports any differences it finds. The dump file can be of type VCD, EVCD or a VPD.

Note:

The vcdiff utility cannot compare dump files of different type.

Dump files consist of two sections:

- A header section that reflects the hierarchy (or some subset) of the design that was used to create the dump file.
- A value change section, which contains all of the value changes (and times when those value changes occurred) for all of the signals referenced in the header.

The vcdiff utility always performs two diffs. First, it compares the header sections and reports any signals/scopes that are present in one dump file but are absent in the other.

The second diff compares the value change sections of the dump files, for signals that appear in both dump files. The

utility determines value change differences based on the final value of the signal in a time step.

The vcdiff Utility Syntax

The syntax of the vcdiff utility is as follows:

```
vcdiff first_dump_file second_dump_file
[-noabsentsig] [-absentsigscope scope] [-absentsigiserror]
[-allabsentsig] [-absentfile filename] [-matchtypes] [-ignorecase]
[-min time] [-max time] [-scope instance] [-level
level_number]
[-include filename] [-ignore filename] [-strobe time1 time2]
[-prestrobe] [-synch signal] [-synch0 signal] [-synch1
signal]
[-when expression] [-xzmatch] [-noxzmatchat0]
[-compare01xz] [-xumatch] [-xdmatch] [-zdmatch] [-zwmatch]
[-showmasters] [-allsigdiffs] [-wrapsize size]
[-limitdiffs number] [-ignorewires] [-ignorereg]
[ingorereals]
[-ignorefunctaskvars] [-ignoretiming units] [-
```

```
ignorestrength]
[-geninclude [filename]] [-spikes]
```

Options for Specifying Scope/Signal Hierarchy

The following options control how the `vcdiff` utility compares the header sections of the dump files:

-noabsentsig

Does not report any signals that are present in one dump file but are absent in the other.

-absentsigscope [scope]

Reports only absent signals in the given scope.

-absentfile [file]

Prints the full path names of all absent scopes/signals to the given file, as opposed to stdout.

-absentsigiserror

If this option is present and there are any absent signals in either dump file, then `vcdiff` returns an error status upon completion even if it doesn't detect any value change differences. If this option is not present, absent signals do not cause an error.

-allabsentsig

Reports all absent signals. If this option is not present, by default, `vcdiff` reports only the first 10 absent signals.

-ignorecase

Ignores the case of scope/signal names when looking for absent signals. In effect, it converts all signal/scope names to uppercase before comparison.

-matchtypes

Reports mismatches in signal data types between the two dump files.

Options for Specifying Scope(s) to be Value Change Diffed

By default, vcdiff compares the value changes for all signals that appear in both dump files. The following options limit value change comparisons to specific scopes.

-scope [scope]

Changes the top-level scope to be value change diffed from the top of the design to the indicated scope. Note, all child scopes/signals of the indicated scope will be diffed unless modified by the **-level** option (below).

-level N

Limits the depth of scope for which value change diffing occurs. For example, if **-level 1** is the only command-line option, then vcdiff diffs the value changes of only the signals in the top-level scope in the dump file.

-include [file]

Reports value change diffs only for those signals/scopes given in the specified file. The file contains a set of full path specifications of signals and/or scopes, one per line.

-ignore [file]

Removes any signals/scopes contained in the given file from value change diffing. The file contains a set of full path specifications of signals and/or scopes, one per line.

Note:

The vcdiff utility applies the `-scope` / `-level` options first. It then applies the `-include` option to the remaining scopes/signals, and finally applies the `-ignore` option.

Options for Specifying When to Perform Value Change Diffing

The following options limit when vcdiff detects value change differences:

`-min time`

Specifies the starting time (in simulation units) when value change diffing is to begin (by default, time 0).

`-max time`

Specifies the stopping time (in simulation units) when value change diffing will end. By default, this occurs at the latest time found in either dump file.

`-strobe first_time delta_time`

Only checks for differences when the strobe is true. The strobe is true at `first_time` (in simulation units) and then every `delta_time` increment thereafter.

`-prestrobe`

Used in conjunction with `-strobe`, tells vcdiff to look for differences just before the strobe is true.

-when *expression*

Reports differences only when the given *when* expression is true. Initially this expression can consist only of scalar signals, combined with and, or, xor, xnor, and not operators and employ parentheses to group these expressions. You must fully specify the complete path (from root) for all signals used in expressions. Note, operators may be either Verilog style (&, |, ^, ~^, ~) or VHDL (and, or, xor, xnor, not).

-synch *signal*

Checks for differences only when the given signal changes value. In effect, the given signal is a "clock" for value change diffing, where diffs are only checked for on transitions (any) of this signal.

-synch0 *signal*

As -sync (above) except that it checks for diffs when the given signal transitions to '0'.

-synch1

As -sync (above) except that it checks for diffs only when the given signal transitions to '1'.

Note:

The -max, -min and -when options must all be true in order for vcdiff to report a value change difference.

Options for Filtering Differences

The following options filter out value change differences that are detected under certain circumstances. For the most part, these options are additive.

-ignoretimings *time*

Ignores the value change when the same signal in one of the VCD files has a different value from the same signal in the other VCD file for less than the specified time. This is to filter out signals that have only slightly different transition times in the two VCD files. The `vcdiff` utility reports a change when there is a transition to a different value in one of the VCD files and then a transition back to a matching value in that same file.

-ignoreregs

Does not report value change differences on signals that are of type register.

-ignorewires

Does not report value change differences on signals that are of type wire.

-ignorereals

Does not report value change differences on signals that are of type real.

-ignorefunctaskvars

Does not report value change differences on signals that are function or task variables.

-ignorestrength (EVCD only)

EVCD files contain a richer set of signal strength and directional information than VCD or even VPD files. This option ignores the strength portion of a signal value when checking for differences.

-compare01xz (EVCD only)

Converts all signal state information to equivalent 4-state values (0, 1, x, z) before difference comparison is made (EVCD files only). Also ignores the strength information.

-xzmatch

Equates x and z values.

-xumatch (9-state VPD file only)

Equates x and u (uninitialized) values.

-xdmatch (9-state VPD file only)

Equates x and d (dontcare) values.

-zdmatch (9-state VPD file only)

Equates z and d (dontcare) values.

-zwmatch (9-state VPD file only)

Equates z and w (weak 1) values. In conjunction with -xzmatch (above), this option causes x and z value to be equated at all times EXCEPT time 0.

Options for Specifying Output Format

The following options change how value change differences are reported.

-allsigdiffs

By default, vcdiff only shows the first difference for a given signal. This option reports all diffs for a signal until the maximum number of diffs is reported (see -limitdiffs).

`-wrapsize` *columns*

Wraps the output of vectors longer than the given size to the next line. By default, this value is 64.

`-showmasters` (VCD, EVCD files only)

Shows collapsed net masters. VCS can split a collapsed net into several sub-nets when this has a performance benefit. This option reports the master signals when the master signals (first signal defined on a net) are different in the two dump files.

`-limitdiffs` *number_of_diffs*

By default, `vcdiff` stops after the first 50 diffs are reported. This option overrides that default. Setting this value to 0 causes `vcdiff` to report all diffs.

`-geninclude` *filename*

Produces a separate file of the given name in addition to the standard `vcdiff` output. This file contains a list of signals that have at least one value change difference. The format of the file is one signal per line. Each signal name is a full path name. You can use this file as input to the `vcat` tool with `vcat`'s `-include` option.

`-spikes`

A spike is defined as a signal that changes multiple times in a single time step. This option annotates with #'s the value change differences detected when the signal spikes (glitches). It keeps and reports a total count of such diffs.

The vcat Utility

The format of a VCD or a EVCD file, although a text file, is written to be read by software and not by human designers. VCS includes the vcat utility to enable you to more easily understand the information contained in a VCD file.

The vcat Utility Syntax

The vcat utility has the following syntax:

```
vcat VCD_filename [-deltaTime] [-raw] [-min time] [-max time]
[-scope instance_name] [-level level_number]
[-include filename] [-ignore filename] [-spikes] [-noalpha]
[-wrapsize size] [-showmasters] [-showdefs] [-showcodes]
[-stdin] [-vgen]
```

Here:

-deltaTime

Specifies writing simulation times as the interval since the last value change rather than the absolute simulation time of the signal transition. Without -deltaTime a vcat output looks like this:

```
--- TEST_top.TEST.U4._G002 ---
0      x
33     0
20000  1
30000  x
30030  z
50030  x
50033  1
60000  0
70000  x
70030  z
```

With `-deltaTime` a vcat output looks like this:

```
--- TEST_top.TEST.U4._G002 ---
0      x
33     0
19967  1
10000  x
30     z
20000  x
3      1
9967   0
10000  x
30     z
```

-raw

Displays “raw” value changed data, organized by simulation time, rather than signal name.

-min *time*

Specifies a start simulation time from which vcat begins to display data.

-max *time*

Specifies an end simulation time up to which vcat displays data.

-scope *instance_name*

Specifies a module instance. The vcat utility displays data for all signals in the instance and all signals hierarchically under this instance.

-level *level_number*

Specifies the number of hierarchical levels for which vcat displays data. The starting point is either the top-level module or the module instance you specify with the **-scope** option.

-include *filename*

Specifies a file that contains a list of module instances and signals. The vcat utility only displays data for these signals or the signals in these module instances.

-ignore *filename*

Specifies a file that contains a list of module instances and signals. However, the vcat utility does NOT display data for these signals or the signals in these module instances.

-spikes

Indicates all zero-time transitions with the >> symbol in the leftmost column. In addition, prints a summary of the total number of spikes seen at the end of the vcat output. The following is an example of the new output:

```
--- DF_test.logic.I_348.N_1 ---
    0      x
    100    0
    120    1
  >>120   0
    4000   1
   12000   0
   20000   1

Spikes detected: 5
```

-noalpha

By default vcat displays signals within a module instance in alphabetical order. This option disables this ordering.

-wrapsize *size*

Specifies value displays for wide vector signals, how many bits to display on a line before wrapping to the next line.

-showmasters

Specifies showing collapsed net masters.

-showdefs

Specifies displaying signals but not their value changes or the simulation time of these value changes.

-showcodes

Specifies displaying the signal's VCD file identifier code.

-stdin

Enables you to use standard input, such as piping the VCD file into vcat, instead of specifying the filename.

-vgen

Generates from a VCD file two types of source files for a module instance: one that models how the design applies stimulus to the instance, and the other that models how the instance applies stimulus to the rest of the design. See “Generating Source Files From VCD Files” on page 7-40.

The following is an example of the output from the vcat utility:

```
vcat exp1.vcd

exp1.vcd: scopes:6 signals:12 value-changes:13

--- top.mid1.in1 ---
0 1

--- top.mid1.in2 ---
0 xxxxxxxx
10000 00000000

--- top.mid1.midr1 ---
0 x
2000 1

--- top.mid1.midr2 ---
0 x
2000 1
```

In this output, for example, you see that signal `top.mid1.midr1` at time 0 had a value of x and at simulation time 2000 (as specified by the `$timescale` section of the VCD file, which VCS derives from the time precision argument of the `'timescale` compiler directive) this signal transitioned to 1.

Generating Source Files From VCD Files

The vcat utility can generate Verilog and VHDL source files that are one of the following:

- A module definition that succinctly models how a module instance is driven by a design, that is, a concise testbench module that instantiates the specified instance and applies stimulus to that instance the way the entire design does. This is called testbench generation.

- A module definition that mimics the behavior of the specified instance to the rest of the design, that is, it has the same output ports as the instance and in this module definition the values from the VCD file are directly assigned to these output ports. This is called module generation.

Note:

The vcat utility can only generate these source files for instances of module definitions that do not have inout ports.

Testbench generation enables you to focus on a module instance, applying the same stimulus as the design does, but at faster simulation because the testbench is far more concise than the entire design. You can substitute module definitions at different levels of abstraction and use vcdiff to compare the results.

Module generation enables you to use much faster simulating “canned” modules for a part of the design to enable the faster simulation of other parts of the design that need investigation.

The name of the generated source file from testbench generation begins with testbench followed by the module and instance names in the hierarchical name of the module instance, separated by underscores. For example `testbench_top_ad1.v`.

Similarly, the name of the generated source file from module generation begins with `moduleGeneration` followed by the module and instance names in the hierarchical name of the module instance, separated by underscores. For example `moduleGeneration_top_ad1.v`.

You enable vcat to generate these files by doing the following:

1. Writing a configuration file.

2. Running vcat with the `-vgen` command-line option.

Writing the Configuration File

The configuration file is named `vgen.cfg` by default and vcat looks for it in the current directory. This file needs three types of information specified in the following order:

1. The hierarchical name of the module instance.
2. Specification of testbench generation with the keyword `testbench` or specification of module generation with the keyword `moduleGeneration`.
3. The module header and the port declarations from the module definition of the module instance.

You can use Verilog comments in the configuration file.

The following is an example of a configuration file:

Example 7-1 Configuration File

```
top.ad1
testbench
//moduleGeneration
module adder (out,in1,in2);
  input in1,in2;
  output [1:0] out;
```

You can use a different name and location for the configuration file. In order to do this, you must enter it as an argument to the `-vgen` option. For example:

```
vcat filename.vcd -vgen /u/design1/vgen2.cfg
```

Example 7-2 Source Code

Consider the following source code:

```
module top;
reg r1,r2;
wire int1,int2;
wire [1:0] result;

initial
begin
$dumpfile("exp3.vcd");
$dumpvars(0,top.pa1,top.ad1);
#0 r1=0;
#10 r2=0;
#10 r1=1;
#10 r2=1;
#10 r1=0;
#10 r2=0;
#10 r1=1;
#10 r2=1;
#10 r1=0;
#10 r2=0;
#10 r1=1;
#10 r2=1;
#10 r1=0;
#10 r2=0;
#100 $finish;
end

passer pa1 (int1,int2,r1,r2);
adder ad1 (result,int1,int2);
endmodule

module passer (out1,out2,in1,in2);
input in1,in2;
output out1,out2;

assign out1=in1;
assign out2=in2;
endmodule
```

```

module adder (out,in1,in2);
  input in1,in2;
  output [1:0] out;

  reg r1,r2;
  reg [1:0] sum;

  always @ (in1 or in2)
  begin
    r1=in1;
    r2=in2;
    sum=r1+r2;
  end

  assign out=sum;
endmodule

```

Notice that the stimulus from the testbench module named `test` propagates through an instance of a module named `passer` before it propagates to an instance of a module named `adder`. The `vcat` utility can generate a testbench module to stimulate the instance of `adder` in the same exact way but in a more concise and therefore faster simulating module.

If we use the sample `vgen.cfg` configuration file in Example 7-1 and enter the following command line:

```
vcat filename.vcd -vgen
```

The generated source file, `testbench_top_ad1.v`, is as follows:

```
module tbench_adder ;
wire [1:0] out ;
reg in2 ;
reg in1 ;
initial #131 $finish;
initial $dumpvars;
initial begin
    #0 in2 = 1'bx;
    #10 in2 = 1'b0;
    #20 in2 = 1'b1;
    #20 in2 = 1'b0;
    #20 in2 = 1'b1;
    #20 in2 = 1'b0;
    #20 in2 = 1'b1;
    #20 in2 = 1'b0;
end
initial begin
    in1 = 1'b0;
    forever #20 in1 = ~in1 ;
end
adder ad1 (out,in1,in2);
endmodule
```

This source file uses significantly less code to apply the same stimulus with the instance of module `passer` omitted.

If we revise the vgen.cfg file to have vcat perform module generation, the generated source file, moduleGeneration_top_ad1.v, is as follows:

```
module adder (out,in1,in2) ;
  input in2 ;
  input in1 ;
  output [1:0] out ;
  reg [1:0] out ;
  initial begin
    #0 out = 2'bxx;
    #10 out = 2'b00;
    #10 out = 2'b01;
    #10 out = 2'b10;
    #10 out = 2'b01;
    #10 out = 2'b00;
    #10 out = 2'b01;
    #10 out = 2'b10;
    #10 out = 2'b01;
    #10 out = 2'b00;
    #10 out = 2'b01;
    #10 out = 2'b10;
    #10 out = 2'b01;
    #10 out = 2'b00;
    #10 out = 2'b01;
    #10 out = 2'b10;
    #10 out = 2'b01;
    #10 out = 2'b00;
  end
endmodule
```

Notice that the input ports are stubbed and the values from the VCD file are assigned directly to the output port.

The vcsplit Utility

The vcsplit utility generates a VCD, EVCD, or VPD file that contains a selected subset of value changes found in a given input VCD, EVCD, or VPD file (the output file has the same type as the input file). You can select the scopes/signals to be included in the generated file either via a command-line argument, or a separate "include" file.

The vcsplit Utility Syntax

The vcsplit utility has the following syntax:

```
vcsplit [-o output_file] [-scope selected_scope_or_signal]
[-include include_file] [-min min_time] [-max max_time]
[-level n] [-ignore ignore_file] input_file [-v] [-h]
```

Here:

-o *output_file*

Specifies the name of the new VCD/EVCD/VPD file to be generated. If *output_file* is not specified, vcsplit creates the file with the default name vcsplit.vcd.

-scope *selected_scope_or_signal*

Specifies a signal or scope whose value changes are to be included in the output file. If a scope name is given, then all signals and sub-scopes in that scope are included.

-include *include_file*

Specifies the name of an include file that contains a list of signals/scopes whose value changes are to be included in the output file.

The include file must contain one scope or signal per line. Each presented scope/signal must be found in the input VCD, EVCD, or VPD file. If the file contains a scope, and separately, also contains a signal in that scope, vcsplit includes all the signals in that scope, and issues a warning.

Note:

If you use both -include and -scope options, vcsplit uses all the signals and scopes indicated.

input_file

Specifies the VCD, EVCD, or VPD file to be used as input.

Note:

If the input file is either VCD or EVCD, and it is not specified, vcsplit takes its input from stdin. The vcsplit utility has this stdin option for VCD and EVCD files so that you can pipe the output of gunzip to this tool. If you try to pipe a VPD file through stdin, vcsplit exits with an error message.

-min *min_time*

Specifies the time to begin the scan.

-max *max_time*

Specifies the time to stop the scan.

-ignore *ignore_file*

Specifies the name of the file that contains a list of signals/scopes whose value changes are to be ignored in the output file.

If you specify neither `include_file` nor `selected_scope_or_signal`, then vcsplit includes all the value changes in the output file except the signals/scopes in the `ignore_file`.

If you specify an `include_file` and/or a `selected_scope_or_signal`, vcsplit includes all value changes of those signals/scopes that are present in the `include_file` and the `selected_scope_or_signal` but absent in `ignore_file` in the output file. If the `ignore_file` contains a scope, vcsplit ignores all the signals and the scopes in this scope.

-level *n*

Reports only *n* levels hierarchy from top or scope. If you specify neither `include_file` nor `selected_scope_or_signal`, `vcsplit` computes *n* from the top level of the design. Otherwise, it computes *n* from the highest scope included.

-v

Displays the current version message.

-h

Displays a help message explaining usage of the `vcsplit` utility.

Note:

In general, any command-line error (such as illegal arguments) that VCS detects causes `vcsplit` to issue an error message and exit with an error status. Specifically:

- If there are any errors in the `-scope` argument or in the `include` file (such as a listing a signal or scope name that does not exist in the input file), VCS issues an error message, and `vcsplit` exits with an error status.
- If VCS detects an error while parsing the input file, it reports an error, and `vcsplit` exits with an error status.
- If you do not provide either a `-scope`, `-include` or `-ignore` option, VCS issues an error message, and `vcsplit` exits with an error status.

Limitations

- MDAs are not supported.
- Bit/part selection for a variable is not supported. If this usage is detected, the vector will be regarded as all bits are specified.

The vcd2vpd Utility

The vcd2vpd utility converts a VCD file generated using \$dumpvars or UCLI dump commands to a VPD file.

The syntax is as shown below:

```
vcd2vpd [-bmin_buffer_size] [-fmax_output_filesize] [-h]
[-m] [-q] [+] [+glitchon] [+nocompress] [+nocurrentvalue]
[+bitrangenospace] [+vpdnoreadopt] [+dut+dut_sufix]
[+tf+tf_sufix] vcd_file vpd_file
```

Usage:

-b<min_buffer_size>

Minimum buffer size in KB used to store Value Change Data before writing it to disk.

-f<max_output_filesize>

Maximum output file size in KB. Wrap around occurs if the specified file size is reached.

-h

Translate hierarchy information only.

-m

Give translation metrics during translation.

-q

Suppress printing of copyright and other informational messages.

+deltacycle

Add delta cycle information to each signal value change.

+glitchon

Add glitch event detection data.

+nocompress

Turn data compression off.

`+nocurrentvalue`

Do not include object's current value at the beginning of each VCB.

`+bitrangenospace`

Support non-standard VCD files that do not have white space between a variable identifier and its bit range.

`+vpdnoreadopt`

Turn off read optimization format.

Options for specifying EVCD options

`+dut+dut_sufix`

Modifies the string identifier for the Device Under Test (DUT) half of the split signal. Default is "DUT".

`+tf+tf_sufix`

Modifies the string identifier for the Test-Fixture half of the split signal. Default is "TF".

`+indexlast`

Appends the bit index of a vector bit as the last element of the name.

`vcd_file`

Specify the vcd filename or use "-" to indicate VCD data to be read from stdin.

`vpd_file`

Specify the VPD file name. You can also specify the path and the filename of the VPD file, otherwise, the VPD file will be generated with the specified name in the current working directory.

The `vpd2vcd` Utility

The `vpd2vcd` utility converts a VPD file generated using the system task `$vcddpluson` or UCLI dump commands to a VCD or EVCD file.

The syntax is as shown below:

```
vpd2vcd [-h] [-q] [-s] [-x] [-xlrn] [+zerodelayglitchfilter]
[+morevhdl] [+start+value] [+end+value] [+splitpacked]
[+dumpports+instance] [-f cmd_filename] vpd_file vcd_file
```

Here:

`-h`

Translate hierarchy information only.

`-q`

Suppress the copyright and other informational messages.

`-s`

Allow sign extension for vectors. Reduces the file size of the generated `vcd_file`.

`-x`

Expand vector variables to full length when displaying `$dumpoff` value blocks.

`-x1rm`

Convert uppercase VHDL objects to lowercase.

`+zerodelayglitchfilter`

Zero delay glitch filtering for multiple value changes within the same time unit.

`+morevhdl`

Translates the VHDL types of both directly mappable and those that are not directly mappable to verilog types.

Note:

This switch may create a non-standard VCD file.

`+start+time`

Translate the value changes starting after the specified start time.

`+end+time`

Translate the value changes ending before the specified end time.

Note:

Specify both start time and end time to translate the value changes occurring between start and end time.

`+dumpports+instance`

Generate an EVCD file for the specified module instance. If the path to the specified instance contains escaped identifiers, then the full path must be enclosed in single quotes.

`-f cmd_filename`

Specify a command file containing commands to limit the design converted to VCD or EVCD. See the “[The Command File Syntax](#)” section for more information.

`+splitpacked`

Use this option to change the way packed structs and arrays are reported in the output VCD file. It does the following:

- Treats a packed structure the same as an unpacked structure and dumps the value changes of each field.

Consider the following example:

```
typedef logic [1:0] t_vec;

typedef struct packed {
    t_vec f_vec_b;
} t_ps_b;

module test();
    t_ps_b var_ps_b;
endmodule
```

The VCD file created in the previous example is as follows:

```
$scope module test $end
$scope fork var_ps_b $end
$var reg 2 ! f_vec_b [1:0] $end
$upscope $end
$upscope $end
```

- Treats a packed MDA as an unpacked MDA except for the inner most dimensions.

Consider the following example:

```
typedef logic [1:0] t_vec;

module test();
    t_vec [3:2] var_vec;
endmodule
```

The VCD file created in the previous example is as follows:

```
$scope module test $end
$var reg      2 %      var_vec[3] [1:0] $end
$var reg      2 &      var_vec[2] [1:0] $end
$upscope $end
```

- Expands all packed arrays defined in a packed struct.

Consider the following example:

```
typedef logic [1:0] t_vec;

typedef struct packed {
    t_vec f_vec;
    t_vec [3:2] [1:0] f_vec_array;
} t_ps;

module test();
    t_ps var_ps;
endmodule
```

The VCD file created in the previous example is as follows:

```
$scope module test $end
$scope fork var_ps $end
$var reg      2 '      f_vec [1:0] $end
$var reg      2 (      f_vec_array[3][1] [1:0] $end
$var reg      2 )      f_vec_array[3][0] [1:0] $end
$var reg      2 *      f_vec_array[2][1] [1:0] $end
$var reg      2 +      f_vec_array[2][0] [1:0] $end
$upscope $end
$upscope $end
```

- Expands all dimensions of a packed array defined in a packed struct.

Consider the following example:

```
typedef logic [1:0] t_vec;

typedef struct packed {
    t_vec f_vec;
    t_vec [3:2][1:0] f_vec_array;
} t_ps;

module test();
    t_ps [1:0] var_paps;
endmodule
```

The VCD file created in the previous example is as follows:

```
$scope module test $end
$scope fork var_paps[1] $end
$var reg      2 '   f_vec [1:0] $end
$var reg      2 (   f_vec_array[3][1] [1:0] $end
$var reg      2 )   f_vec_array[3][0] [1:0] $end
$var reg      2 *   f_vec_array[2][1] [1:0] $end
$var reg      2 +   f_vec_array[2][0] [1:0] $end
$upscope $end
$scope fork var_paps[0] $end
$var reg      2 ,   f_vec [1:0] $end
$var reg      2 -   f_vec_array[3][1] [1:0] $end
$var reg      2 .   f_vec_array[3][0] [1:0] $end
$var reg      2 /   f_vec_array[2][1] [1:0] $end
$var reg      2 0   f_vec_array[2][0] [1:0] $end
$upscope $end
$upscope $end
```

- Expands and prints the value of each member of a packed union.

Consider the following example:

```
module testit;

    typedef logic [1:0] t_vec;

    typedef union packed {
        t_vec f_vec;
        struct packed {
            logic f_a;
            logic f_b;
        } f_ps;
    } t_pu_v;
    typedef union packed {
        struct packed {
            logic f_a;
            logic f_b;
        } f_ps;
        t_vec f_vec;
    } t_pu_s;
    t_pu_v var_pu_v;
    t_pu_s var_pu_s;
endmodule
```

The VCD file created in the previous example is as follows:

```
$scope module testit $end
$scope fork var_pu_v $end
$var reg      2 -      f_vec [1:0] $end
$scope fork f_ps $end
$var reg      1 .      f_a $end
$var reg      1 /      f_b $end
$upscope $end
$upscope $end
$scope fork var_pu_s $end
$scope fork f_ps $end
$var reg      1 0      f_a $end
$var reg      1 1      f_b $end
```

```
$upscope $end
$var reg          2 2      f_vec [1:0] $end
$upscope $end
$upscope $end
```

The Command File Syntax

Using a command file, you can generate:

- A VCD file for the whole design or for the specified instances.
- Only the port information for the specified instances.
- An EVCD file for the specified instances.

Note the following before writing a command file:

- All commands must start as the first word in the line, and the arguments for these commands should be written in the same line. For example:

```
dumpvars 1 adder4
```

- All comments must start with “//”. For example:

```
//Add your comment here
dumpvars 1 adder4
```

- All comments written after a command, must be preceded by a space. For example:

```
dumpvars 1 adder4 //can write your comment here
```

A command file can contain the following commands:

```
dumpports instance [instance1 instance2 ...]
```

Specify an instance for which an EVCD file has to be generated. You can generate an EVCD file for more than one instance by specifying the instance names separated by a space. You can also specify multiple `dumpports` commands in the same command file.

```
dumpvars [level] [instance instance1 instance2 ...]
```

Specify an instance for which a VCD file has to be generated. [*level*] is a numeric value indicating the number of levels to traverse down the specified instance. If not specified, or if the value specified is "0", then all the instances under the specified instance will be dumped.

You can generate a VCD file for more than one instance by specifying the instance names separated by a space. You can also specify multiple `dumpvars` commands in the same command file.

If this command is not specified or the command has no arguments, then a VCD file will be generated for the whole design.

```
dumpvcdports [level] instance [instance1 instance2  
....]
```

Specify an instance whose port values are dumped to a VCD file. [level] is a numeric value indicating the number of levels to traverse down the specified instance. If not specified, or if the value specified is "0", then the port values of all the instances under the specified instance will be dumped.

You can generate a dump file for more than one instance by specifying the instance names separated by a space. You can also specify multiple `dumpvcdports` commands in the same command file.

Note:

`dumpvcdports` splits the inout ports of type wire into two separate variables:

- one shows the value change information driven into the port. VCS adds a suffix `_DUT` to the basename of this variable.
- the other variable shows the value change information driven out of the port. VCS adds a suffix `_TB` to the basename of this variable.

`dutsuffix DUT_suffix`

Specify a string to change the suffix added to the variable name that shows the value change date driven out of the inout port. The default value is `_DUT`. The suffix can also be enclosed within double quotes.

`tbsuffix TB_suffix`

Specify a string to change the suffix added to the variable name that shows the value change date driven into the inout port. The default value is `_TB`. The suffix can also be enclosed within double quotes.

`starttime start_time`

Specify the start time to start dumping the value change data to the VCD file. If this command is not specified, the start time will be the start time of the VPD file.

Note:

Only one `+start` command is allowed in a command file.

`endtime end_time`

Specify the end time to stop dumping the value change data to the VCD file. If this command is not specified, the end time will be the end time of the VPD file.

Note:

Only one `+end` command is allowed in a command file, and must be equal to or greater than the start time.

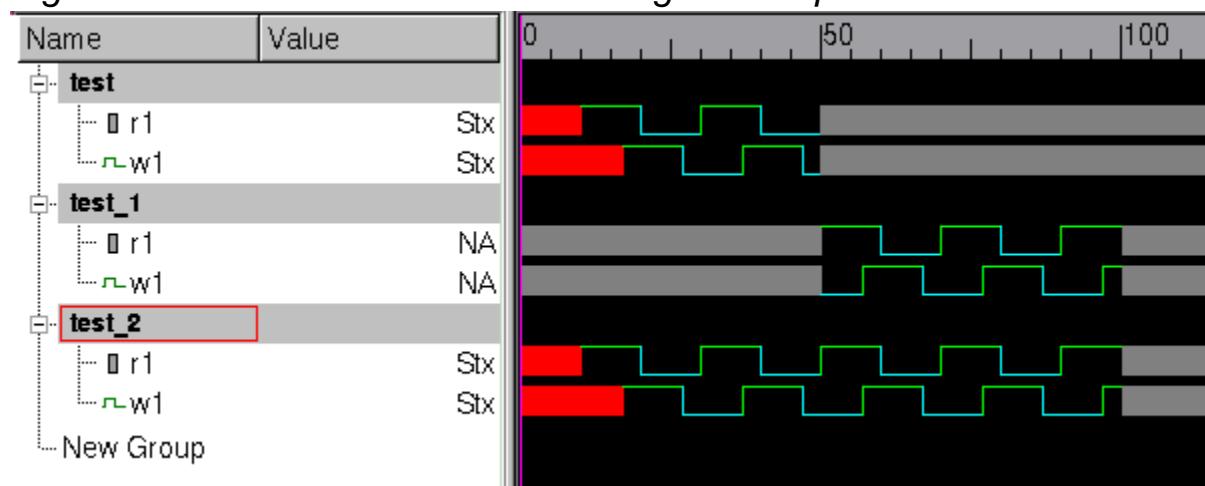
Limitations

- `dumpports` is mutually exclusive with either the `dumpvars` or `dumpvcdports` commands. The reason for this is that `dumpports` generates an EVCD file while both `dumpvars` and `dumpvcdports` generates standard VCD files.
- Escaped identifiers must include the trailing space.
- Any error parsing the file will cause the translation to terminate.

The vpdmerge Utility

Using the vpdmerge utility, you can merge different VPD files storing simulation history data for different simulation times, or parts of the design hierarchy into one large VPD file. For example in the DVE Wave Window in [Figure 7-6](#), there are three signal groups for the same signals in different VPD files.

Figure 7-6 DVE Wave Window with Signal Groups from Different VPD Files



Signal group `test` is from a VPD file from the first half of a simulation, signal group `test_1` is from a VPD file for the second half of a simulation, and signal group `test_2` is from the merged VPD file.

The syntax is as shown below:

```
vpdmerge [-h] [-q] [-hier] [-v] -o merged_VPD_filename  
input_VPD_filename input_VPD_filename ...
```

Usage:

`-h`

Displays a list of the valid options and their purpose.

-o *merged_VPD_filenames*

Specifies the name of the output merged VPD file. This option is required.

-q

Specifies quiet mode, disables the display of most output to the terminal.

-hier

Specifies that you are merging VPD files for different parts of the design, instead of the default condition, without this option, which is merging VPD files from different simulation times.

-v

Specifies verbose mode, enables the display of warning and error messages.

Restrictions

The vpdmerge utility includes the following restrictions:

- To read the merged VPD file, DVE must have the same or later version than that of the vpdmerge utility.
- VCS must have written the input VPD files on the same platform as the vpdmerge utility.
- The input VPD files cannot contain delta cycle data (different values for a signal during the same time step).
- The input VPD files cannot contain named events.
- The merged line stepping data does not always accurately replay scope changes within a time step.

- If you are merging VPD files from different parts of the design, using the `-hier` option, the VPD files must be used for distinctly different parts of the design, they cannot contain information for the same scope.

Limitations

The verbose option `-v` may not display error or warning messages in the following scenarios:

- If the reference signal completely or coincidentally overlaps the compared signal.
- During hierarchy merging, if the design object already exists in the merged file.

During hierarchy merging, the `-heir` option may not display error or warning messages in the following scenarios.

- If the start and end times of the two dump files are the same.
- If the datatype of the hierarchical signal in the dump files do not match.

Value Conflicts

If the `vpdmerge` utility encounters conflicting values for the same signal, with the same hierarchical name, in different input VPD files, it does the following when writing the merged VPD file:

- If the signals have the same end time, `vpdmerge` uses the values from the first input VPD file that you entered on the command line.
- If the signals have different end times, `vpdmerge` uses the values for the signal with the greatest end time.

In cases where there are value conflicts, the `-v` option displays messages about these conflicts.

8

Using Discovery Visual Environment

This chapter describes the basic use of the Discovery Visual Environment (DVE) to debug your design. It includes the following topics:

- “General Requirements”
- “DVE Options”
- “Working with the DVE”
- “Running a Simulation”

General Requirements

You must use the same version of VCS and DVE to ensure problem-free debugging of your simulation. To check the DVE version:

- Enter the `dve -v | v` command-line option.
- Use the **Help > About** menu option.

DVE Options

This section describes DVE options and requirements.

Help Options

The DVE command-line help options are:

`help`

Displays DVE basic commands.

`help -all`

Displays all DVE commands.

`dve -v | -V`

Displays version information.

Post-processing, Interactive, and Script Options

`dve`

With no arguments, displays an empty DVE TopLevel window.
DVE usage can be post-processing or interactive mode from this point.

`dve -vpd filename`

Opens up DVE, reads VPD file given on the command line, and opens the top-level scope for that design.

`dve -vpd filename -session filename`

Opens up DVE and reads VPD file given on the command line, then opens a previously saved session TCL file.

`simv -gui`

Opens DVE with the simv simulator attached at time 0.

`vcs -gui -R`

Same as above, but invoked at compile time.

`dve -cmd "cmd"`

Starts DVE and executes the Tcl command enclosed in quotation marks. Multiple commands separated by semicolons are allowed.

`dve -script name`

Starts DVE and reads in a Tcl script specified by *name*.

```
dve -session name
```

Starts DVE and reads in a session file. If the `-session` and `-script` options are combined, the session is read first and then the script.

64-bit Platform Support

```
-full64
```

Enables 64-bit DVE functionality when entered at runtime when you have specified the platform. To activate 64-bit support enter:

```
dve -full64
```

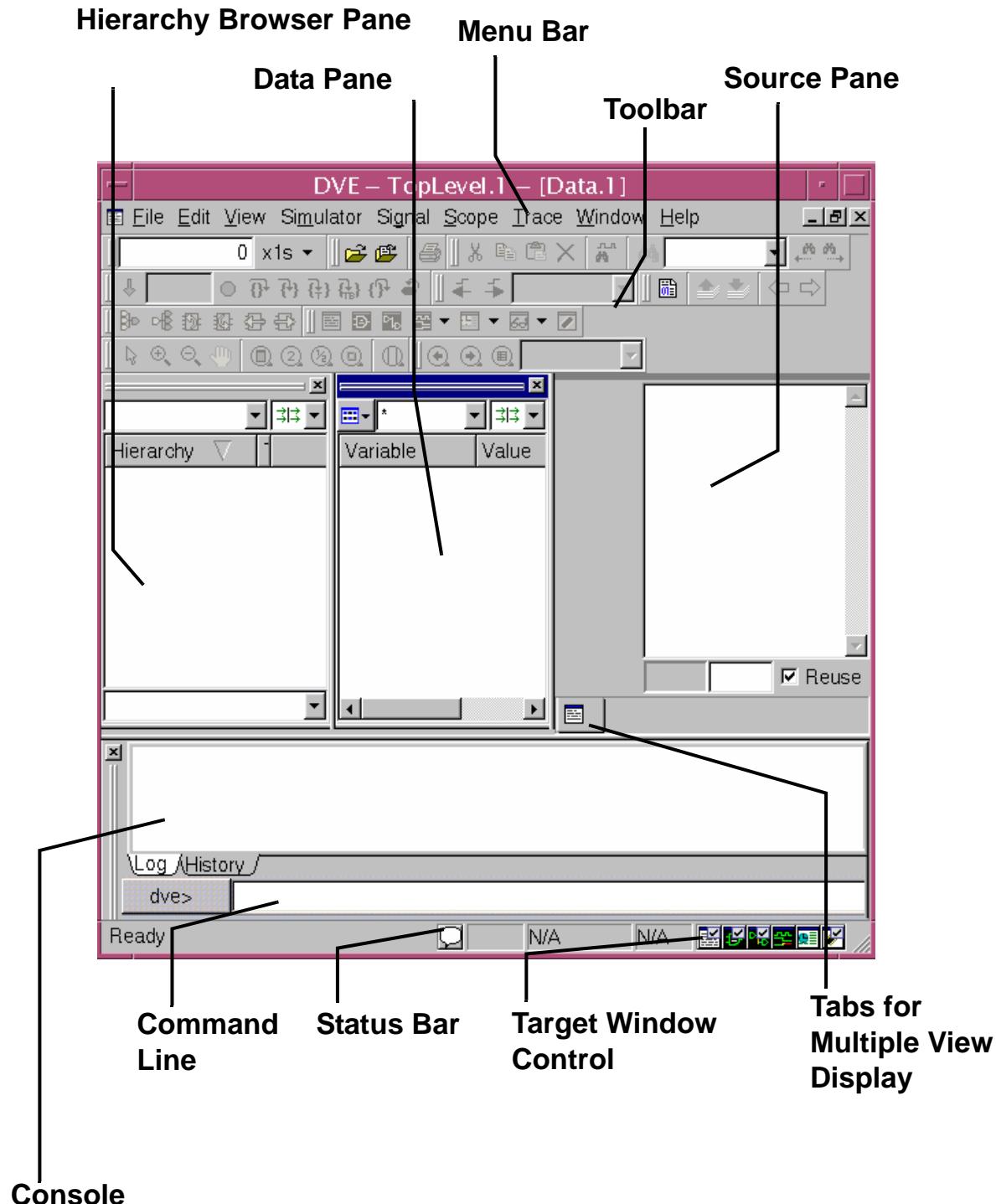
Working with the DVE

DVE has a completely flexible window model. This model is based on the concept of the TopLevel window.

A TopLevel window contains a menu, toolbar, status bar, and pane targets. Any number of TopLevel windows are possible. The default at startup is one window.

A DVE TopLevel window is a frame for displaying design and debug data. The default DVE window configuration is to display the TopLevel window with the Hierarchy Browser on the left, the Console pane at bottom, and the Source window occupying the remaining space. You can change the default using the preference file, the session file or a startup script. [Figure 8-1](#) shows the default TopLevel window.

Figure 8-1 DVE TopLevel Window Initial View



A TopLevel window is a frame that displays panes and views.

- A pane can be displayed one time on each TopLevel Window and it serves a specific debug purpose. Examples of panes are Hierarchy, Data, and the Console panes.
- A view can have multiple instances per TopLevel window. Examples of views are Source, Wave, List, Memory, and Schematic.

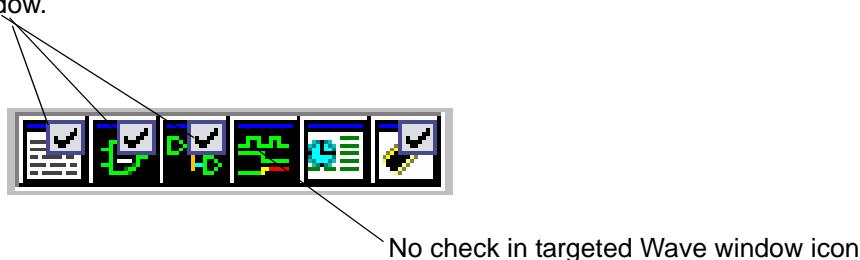
Panes can be docked on any side of a TopLevel window or can be left floating in the area in the frame not occupied by docked panes (called the workspace).

Managing Target Views

The target policy dictates where views will be created. You can display multiple views per TopLevel window. On each TopLevel window at the bottom right corner of the frame are target icons ([Figure 8-2](#)). These icons represent view types.

Figure 8-2 Window targeting icons

Check marks indicate targeted windows are attached to the current window.



Target icons can have the following two states:

- Targeted – Icon has a check mark in it, which means an action that requires a new view creates that view in the current TopLevel window.
 - Untargeted – icon has no dart in it, which means an action that requires a new view places the view in the most recently displayed TopLevel window, or creates a new TopLevel window that contains that view when no other TopLevel window exists.
-

Populating Other Views and Panes

Use the Hierarchy pane to view data in other DVE views and panes.

Displaying Variables in the Data Pane

Single-clicking on a hierarchy object selects it. Selecting a hierarchy object automatically displays that object's variables in the data pane.

You can drag and drop a selected object into any other DVE pane or view that will accept it (such as the Source view, the Wave view, and the List view).

Dragging and Dropping Scopes

A selected object can be dragged and dropped into any other DVE pane or view that will accept it. You can do the following scope dragging operations:

- Dropping a scope into the Source view displays the definition of that object in the source view and selects the definition line.

- Dropping a scope into the Data Pane causes the scope to be selected in the Hierarchy pane and displays the scope variables in the Data pane.

If the scope is already selected in the Hierarchy pane, dropping it in the Data pane has no effect because the variables of the scope selected in the Hierarchy pane are automatically displayed in the Data pane.

- Dropping a scope into the Wave view adds all the scopes signals to a new group or puts them under the insertion bar of the wave signal list.
- Dropping a scope into the Schematic view displays the design schematic for that scope.
- Dropping a scope into the Path Schematic view has no useful results.
- Dropping a scope into the Memory view is not allowed.
- Dropping a scope from elsewhere into the Hierarchy pane view selects that scope, if it is available.
- If you drag a scope into the Hierarchy pane but the object has not yet been loaded into the Hierarchy pane, use the **Edit->Search For Signal/Scopes** menu option.

- Dropping a scope into a text area such as the DVE command line drops the full hierarchical text. The exception to this is dropping a scope in the Find Dialog text entry area (either dialog or toolbar area). In this special case, just the leaf string is dropped. For example, dropping `top.c.b.a` results in just `a` in the Find text area.

If you select more than one hierarchy object (you can do this with Ctrl-Left Click), the object closest to the linear top of the list is dropped. For example:

```
top
  top.a
    top.a.b
  top.b
    top.b.b
```

In this example, if you select, drag and drop both `top.a.b` and `top.b` into a text area, DVE drops only `top.a.b`.

- Double-clicking on a Hierarchy object causes the object's definition to be displayed and highlighted in the Source view (based on the reuse policy of the Source view).

Using the Menu Bar and Toolbar

Complete information on DVE menu and toolbar is available in the DVE help.

1. Select **Help > DVE Help**.
2. From the main help screen, select **Using the Graphical User Interface**, then select **Using the Menu Bar and Toolbar**.

This section displays definitions for all commands.

Running a Simulation

You can open DVE and start the simulation from the gui.

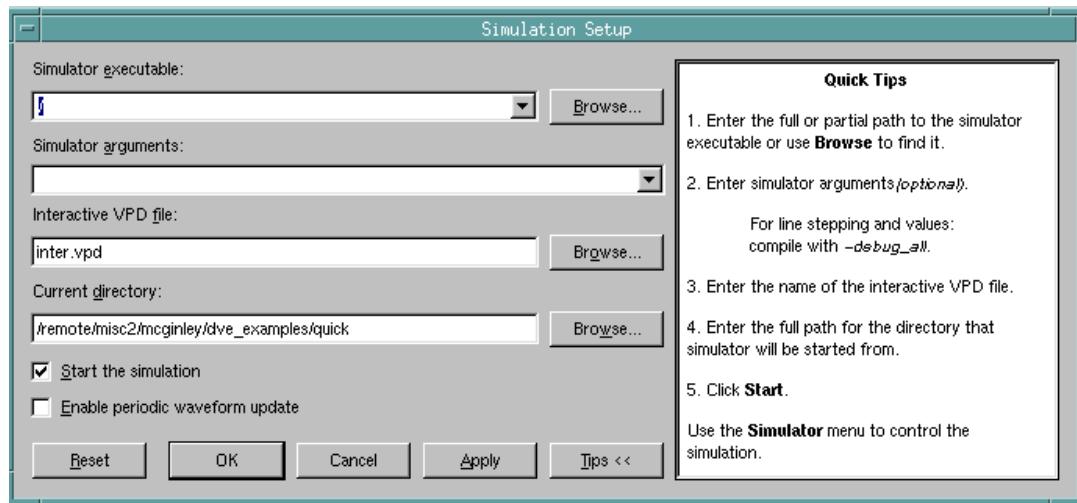
Setting Up and Starting an Interactive Session

In addition to loading VPD files for post-processing, you can also setup and run a simulation interactively in real-time using a compiled Verilog design.

1. From the command line, open DVE.

```
%dve
```

2. Select **Simulator > Setup**, then click the **Tips** button..



3. Follow the procedure to start the simulation.

Note:

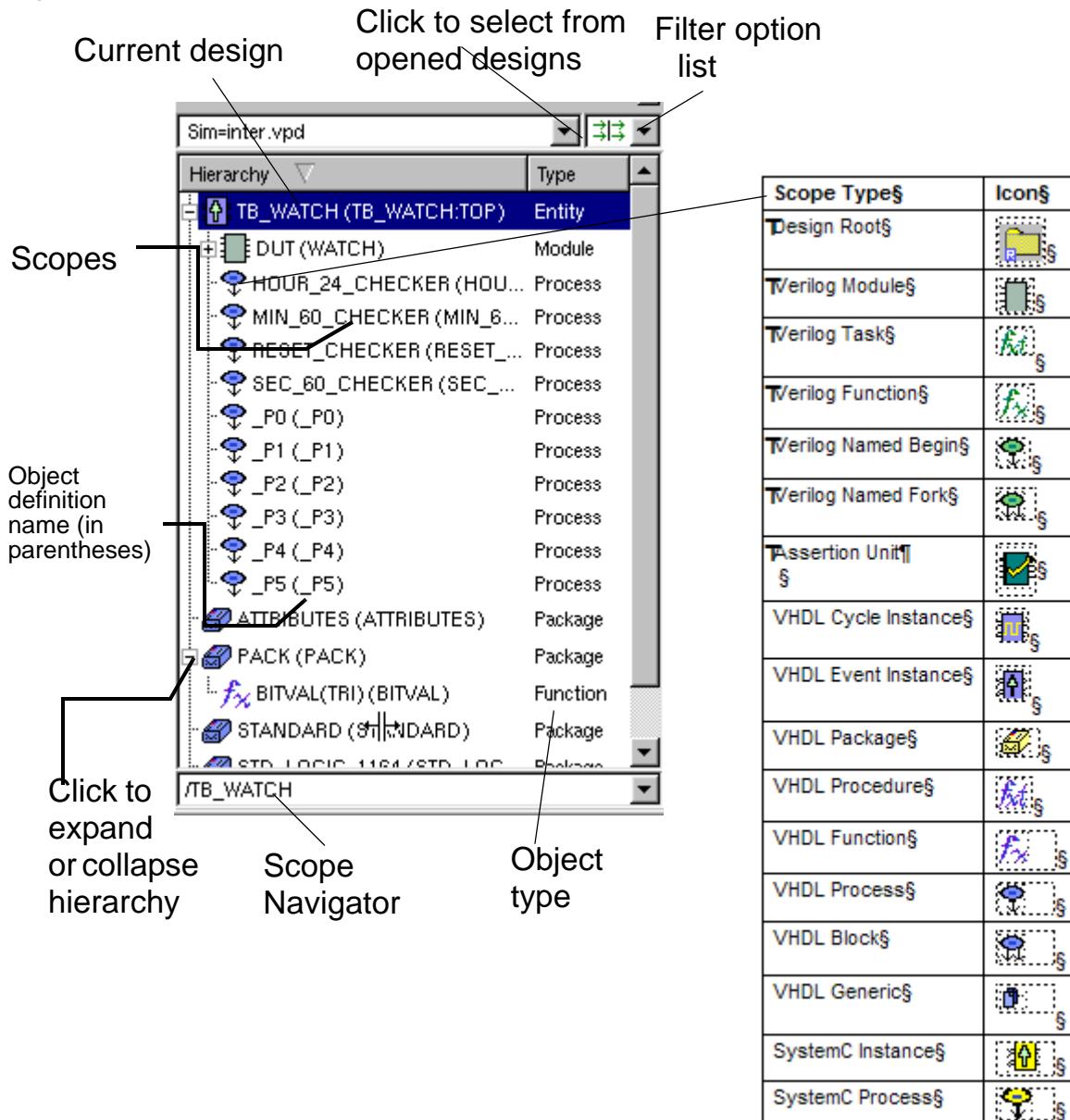
You can display instructions for any dialog box task by clicking on the **Tips** button.

Navigating the Design Hierarchy

The Hierarchy pane, shown in [Figure 8-3](#), is composed of two drop-down lists on top of a tree view.

- The drop-down list on the left is the design selection list. It contains a list of currently open designs with the current design at the top.
- The drop-down list on the right is for filtering object types.
- The tree view is made up of two columns: Hierarchy and Type.
 - The Hierarchy column shows the static instance tree. The names in the instance tree are in the instance name (definition name) format. Top modules (or scopes) are at the top-level of the tree.
 - The type column displays the type of hierarchical object.
- User the Scope Navigator to display a signal in the hierarchy. Enter a full signal path. The hierarchy list expands to display the signal.
- Right-click to display the context-sensitive menu.

Figure 8-3 Hierarchy Pane



Working with Scopes and Signals

VCS displays simulation analysis data corresponding to the contents of the scope you select in the Hierarchy pane.

To select a scope in the Hierarchy pane:

- Click on the scope name to select the scope and populate the Data pane.
- Double-click anywhere on the name of the scope to select the scope and populate the Data pane and the Source View. See [Figure 8-4](#).

Viewing Values

The Data pane displays values for displayed signals at the current simulation time. Select an object in the Hierarchy pane to display the signals in the Data pane.

Viewing Signal Source Code

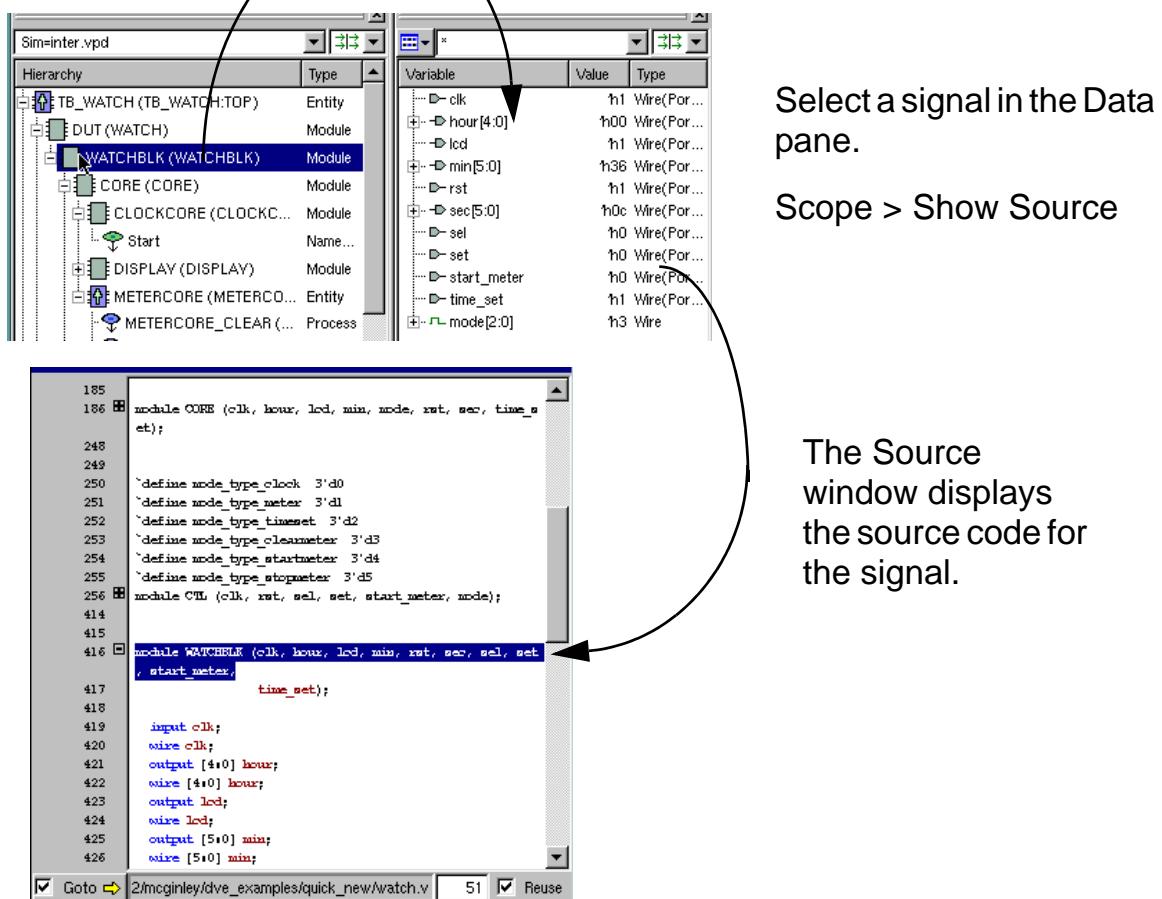
To view source code for a signal in the Data pane, select a signal in the Data pane, then select **Source > Show Source**.

The Source view displays the source code for the selected signal. See [Figure 8-4](#) for an illustration of the process.

Figure 8-4 Display of Data for a Scope

Select a scope in the Hierarchy pane to populate the Data Pane.

The Data Pane displays signals in the selected scope.



Dumping Signal Values

There are two ways you can specify scope and signal values to save:

- Select signals in the hierarchy pane of the TopLevel View or the tree view of the Wave or List view, then select **Simulator > Dump** or right-click in the Hierarchy Pane and select **Dump**.

- Using the Dump Values dialog box.

Select **Simulator > Add Dump** or right-click in the Hierarchy Pane and select **Add Dump**. In the Dump Values dialog box, specify the scopes and signals and click **Dump**.

Forcing Signal Values

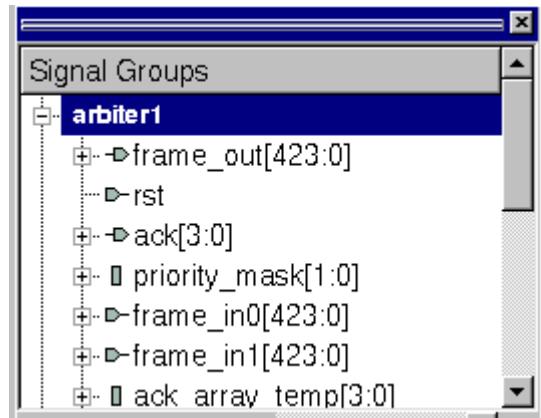
You can force a value onto a signal or variable by specifying force criteria in the Force Values dialog box.

1. Select **Simulator > Add Force** to display the Add Force dialog box.
2. Click the **Tips** button, to display Help, then follow the instructions.
3. Then, click on one of the following:
 - **OK** to apply your force specification and close the dialog box.
 - **Apply** to apply your force specification and have the dialog box remain open.
 - **Cancel** to close and not apply the specification.

Working with Signal Groups

The Signal Groups pane displays signals in groups based on the design. You can create and modify user-defined groups containing signals of interest and view multiple signal groups simultaneously in a Wave view. Click the plus sign  to see the contents and click the minus sign  to hide the contents of a group.

Figure 8-5 Signal Group pane



Creating Signal Groups

To quickly create a new user-defined signal group:

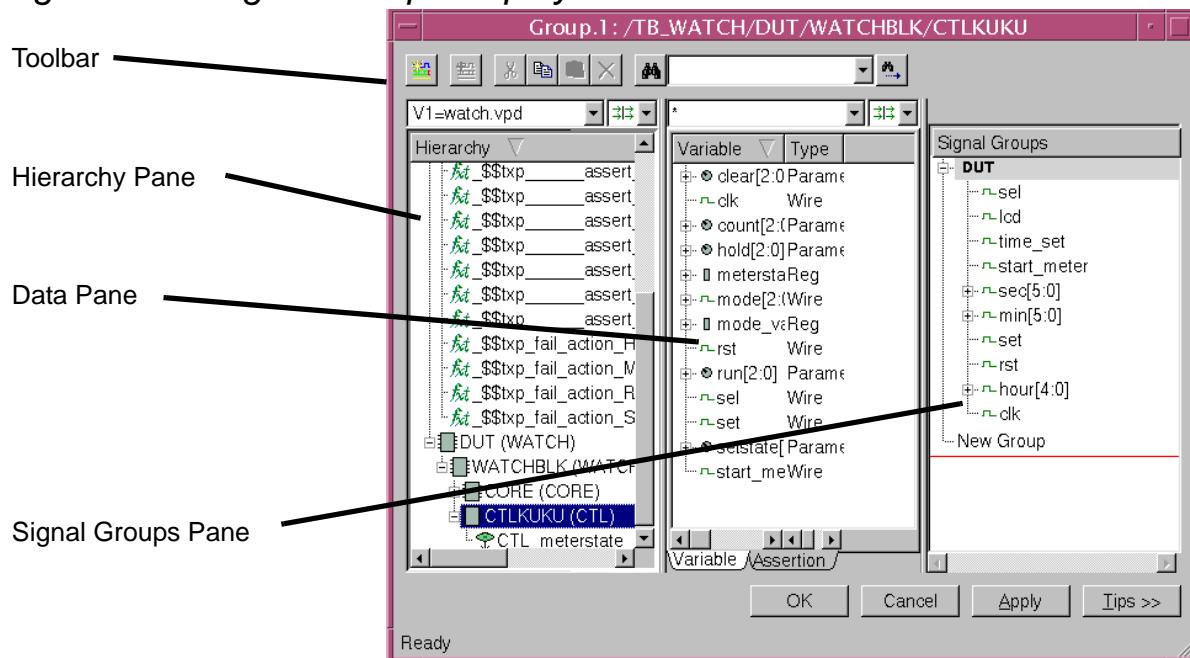
1. Double-click **New Group** in the Signal Pane or select **Signal > Add to Group**.
2. Drag and drop or copy and paste signals from the Signal Pane or another DVE pane, such as the Data Pane of a TopLevel Window, to the Wave view Signal pane.

Managing Signal Groups

You can use the Signal Groups pane and the Wave view Signal pane to create, delete, modify, and search Signal Groups.

The Signal Groups pane displays the selected grouped signals as shown in [Figure 8-6](#).

Figure 8-6 Signal Groups display



In the Wave view Signal Pane or Signal Groups pane:

- Click **Create Group** to create a new group.
- Use the Hierarchy Pane to navigate the design and display signals in the Data Pane.
- Select a signal or signals in the data pane or Wave view Signal pane and click **Add to Groups**, drag and drop, or **Copy/Paste** to include the signal or signals in a selected group.

- Select a signal group or select signals from a group, then click Delete  to remove the item.
- Search for a signal group, string, or expression using the toolbar Search  buttons.

Displaying Signal Groups

To control the display of signal groups, select **Signal > Display Signal Group**, then select and deselect signal groups to display and hide.

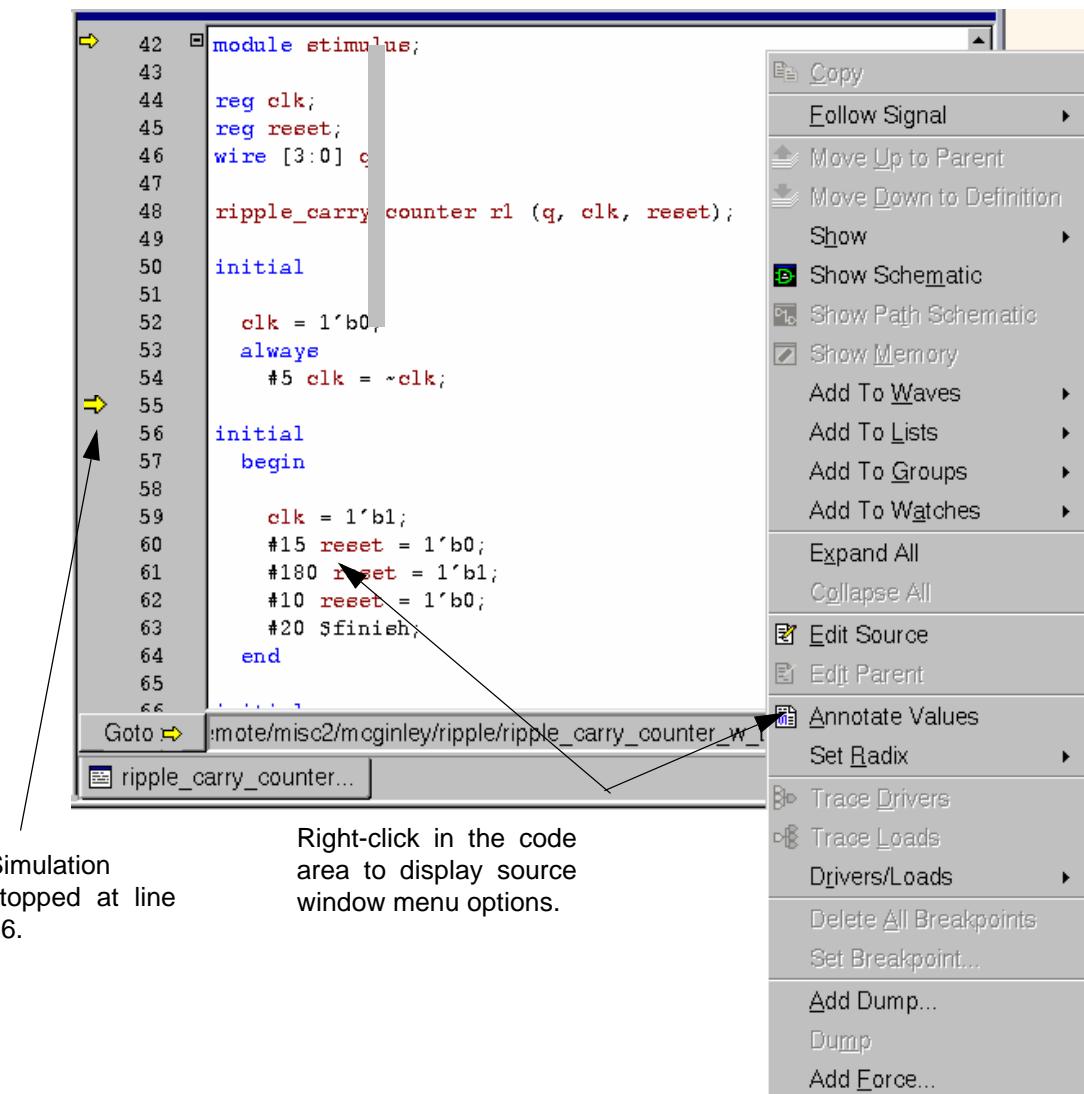
Merging Signal Groups

Merge signal groups by selecting one or more signal groups in the Signal Group Pane, then dragging and dropping the selection into another signal group.

Working with the Source Code

Use the Source view to navigate through the design and view results in other DVE windows by dragging and dropping signals, scopes, and objects into other windows or right-clicking and selecting a menu command as shown in the following illustration.

Figure 8-7 Source view context-sensitive menu



Managing Breakpoints in Interactive Simulation

DVE allows setting breakpoints that cause the tool to stop when stepping or running during interactive simulation:

- Line breakpoints execute each time a specified line is reached during simulation about line breakpoints. You can also specify an instance to have the tool stop only at the line in the specified instance.
- Time breakpoints stop at a specified absolute or relative time in the simulation.
- Signal breakpoints trigger when a specified signal rises, falls, or changes.
- Assertion breakpoints stop at a specified assertion event.
- Task/Function breakpoints stop at the specified task or function.

Controlling Line Breakpoints from the Source View

You can control line breakpoints in the Source view attribute area using your mouse.

- To set a breakpoint, left-click in the attributes area of the Source view next to an executable line. A solid red circle displays to indicate a line breakpoint is set.

Note:

A line breakpoint can only be set on an executable line. If a line is not executable, no breakpoint will be set when you right-click next to it.

- Right-click in the attributes area of the Source view, then select Set Breakpoint from the context-sensitive menu. A solid red circle displays to indicate a line breakpoint is set.
- To disable a line breakpoint, left-click on the solid red breakpoint circle. The circle is no longer solid, indicating the breakpoint is disabled.

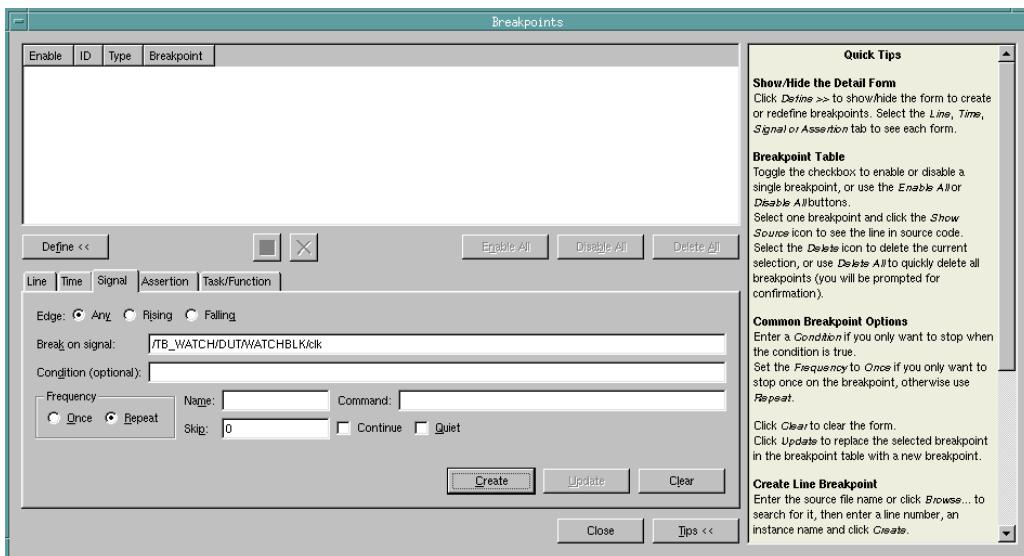
- Right-click on an enabled or disabled breakpoint, then select **Disable Breakpoint** or **Enable Breakpoint**, **Delete Breakpoint**, or **Delete All Breakpoints**. You can also delete all breakpoints from anywhere in the attribute area.

Creating Breakpoints from the Dialog Box

You can create the following types of breakpoints in an interactive simulation from the Breakpoints dialog box:

- Right-click in the Source view line attribute area or pull down the Simulator menu, then select **Set Breakpoints** to display the Breakpoints dialog box (Figure 8-8).
- Select the tab labeled with the desired type of breakpoint.
- Click the **Tips** button, to display Help, then follow the instructions.

Figure 8-8 Creating Breakpoints



- Click **Create**.

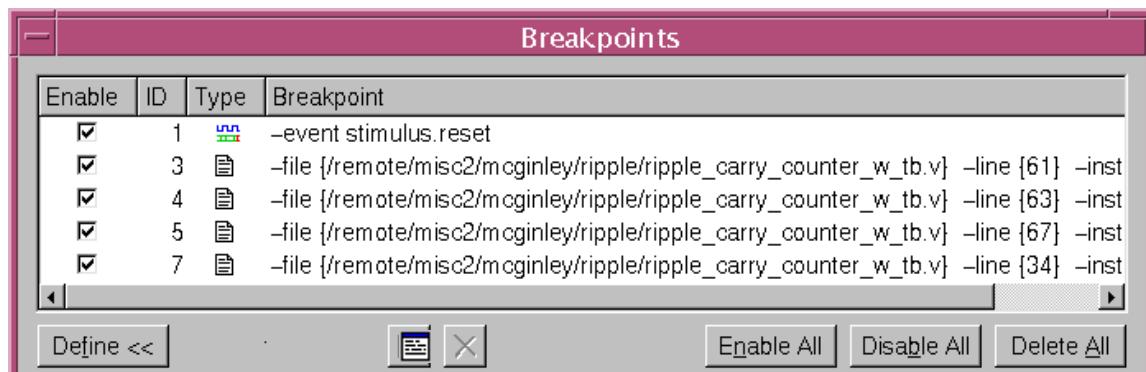
The breakpoint is created and appears in the breakpoint list box.

Edit Breakpoints

To edit a breakpoint:

1. In the Source view line attribute area or from the Simulator menu, right-click and select **Set Breakpoints** to display the Breakpoints dialog box ([Figure 8-9](#)).

Figure 8-9 Breakpoints dialog box



2. To work with defined breakpoints, perform one of the following:
 - To enable or disable a breakpoint, select or deselect its Enable box or select the breakpoint in the list and click **Enable All** or **Disable All**.
 - To delete a breakpoint, select the breakpoint in the list and click or click **Delete All** to delete all breakpoints.
 - To view the source code of a breakpoint, click .

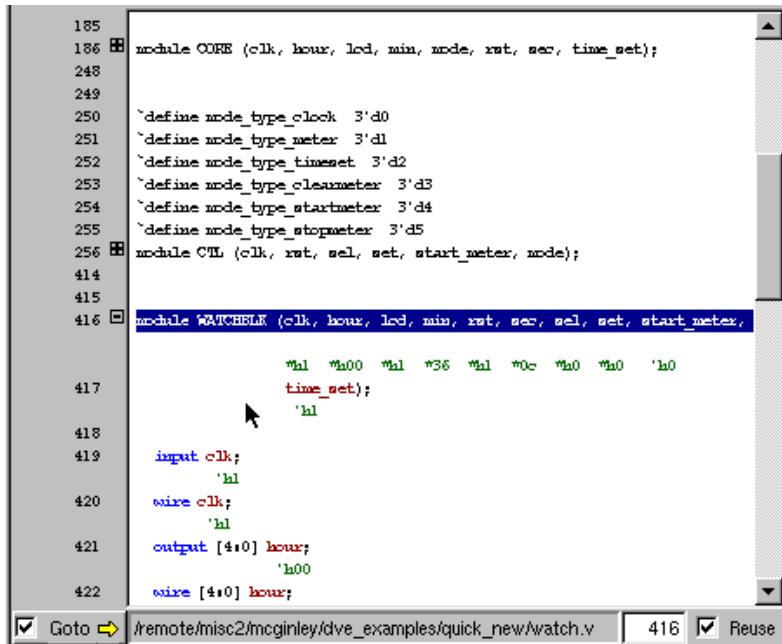
Annotating Values

To display annotated values for a selected block in the Source view, do the following:

Value annotation for variables/signals in the Source view can be enabled in three ways:

- Click the Annotate Values Icon  in the tool bar.
- Select **Scope->Annotate Values**.
- In the Source view, right click, then Select **Annotate Values**.

Figure 8-10 *Values annotated in the Source view*



The screenshot shows a portion of a Verilog source code file. Lines 185 through 422 are visible. The code defines several modules: CORE, CLK, and WATCHER. The WATCHER module has inputs clk, hour, lcd, min, rot, sec, sel, set, start_meter, and time_set. It also has an input clk and an output hour. The annotations show the current value of each signal: clk is 'h1, hour is 'h00, lcd is 'h00, min is 'h36, rot is 'h0c, sec is 'h00, sel is 'h0, set is 'h0, start_meter is 'h0, and time_set is 'h1. A cursor is positioned over the 'h1 annotation for time_set. The status bar at the bottom shows the path /remote/misc2/mcgirney/dve_examples/quick_new/watch.v, line 416, and a checked 'Reuse' checkbox.

Using Waveforms

The Wave view displays waveforms for signals, traced assertions, and signal comparison. You can also create notations of times of interest using named markers or cursors.

Loading Waveform Views

To view waveform information for signals in the Wave view:

- Select a scope or object of interest from the Hierarchy Browser, Variable pane, Source view, List view, Schematic view, or Assertion pane, then click the **Add to Waves** icon in the toolbar .

You can either add the selected signals to the new waveform view or to the recently used waveform view. Just clicking on this icon will add signals to the recently used waveform view.

- In the Hierarchy Browser and Variable pane, you can also right-click and select **Add to Waves** from the context-sensitive menu.

Using the Signal Pane

The Signal Pane displays signals in groups:

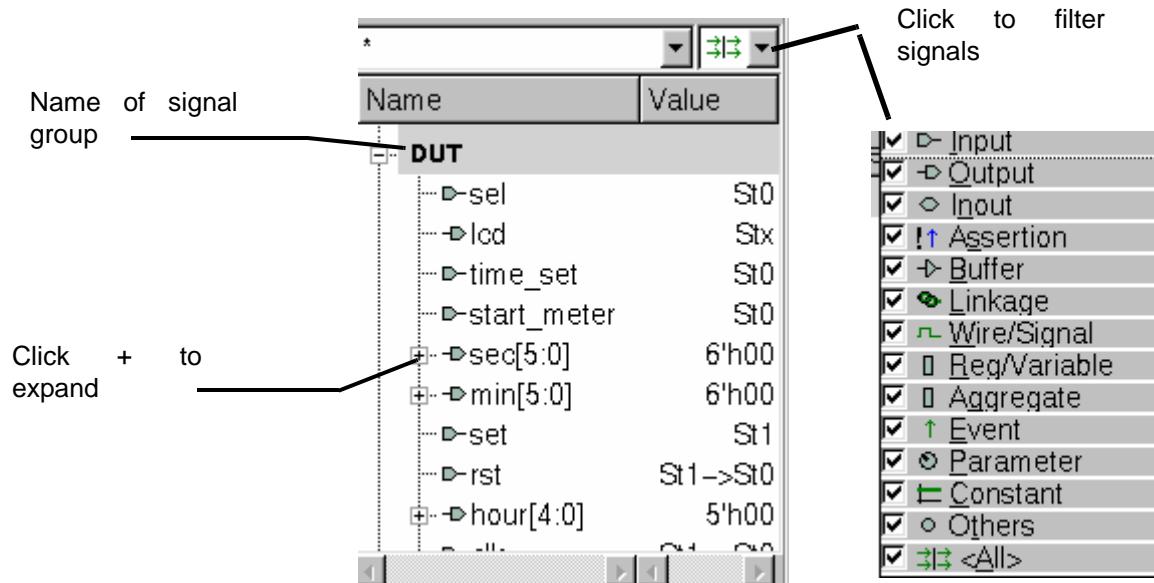
- Scalar signals have their value displayed in binary radix.
- Vector signals have their values displayed in hexadecimal radix.
- Integers, real numbers, and times are displayed in the floating point radix.

The components of the Signal Pane are:

- The header allows filtering of signals displayed in the Waves view.
- The **Name** column displays signal names.
- The **Value** column displays the value of signals at the simulation time selected by the C1 cursor (which is also the value in the TopLevel Window Time field).

See [Figure 8-11](#) for an example of the Signal Pane.

Figure 8-11 The Signal Pane



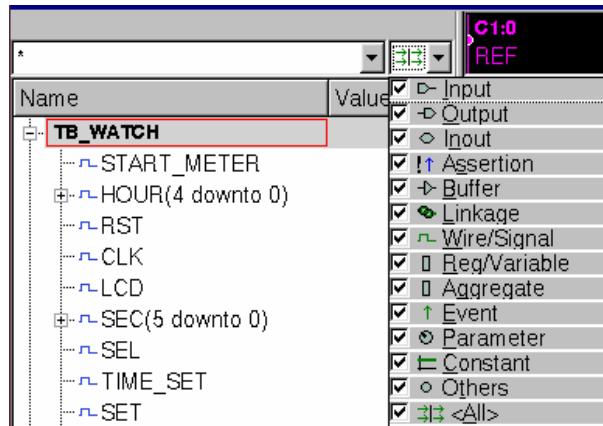
Renaming Signal Groups

To rename a signal group, left-click on the signal in the Hierarchy pane to select the signal group, then left-click again to select the name. Enter the new signal name.

Filtering Signals

Use the pull-down menu in the upper right of the Signal Pane to filter signal display in the Signal Pane. Deselect a signal to hide its display; select the check box to redisplay it.

Figure 8-12 Filter signal display menu



Adding Signal Dividers

To separate signals in the Wave view, you can add dividers between signals by selecting **Signal > Insert Divider**. You can add as many dividers between signals as you like.

A divider inserted into a Signal Group displays in all instances of that signal group opened in Wave views. Dividers are saved in the session TCL file and are restored when the session is opened.

Customizing Duplicate Signal Display

When displaying duplicate signals, you can customize the display of an instance of a signal without affecting the display of any duplicates.

1. Select a signal, then toggle **Signal > Default Properties** off.
2. Select **Signal > Properties** and make any changes to the signal scheme, or color.

Note:

You can also right-click and select **Properties** from the context-sensitive menu.

Changes are made to the selected signal without affecting the display of duplicate signals.

Note:

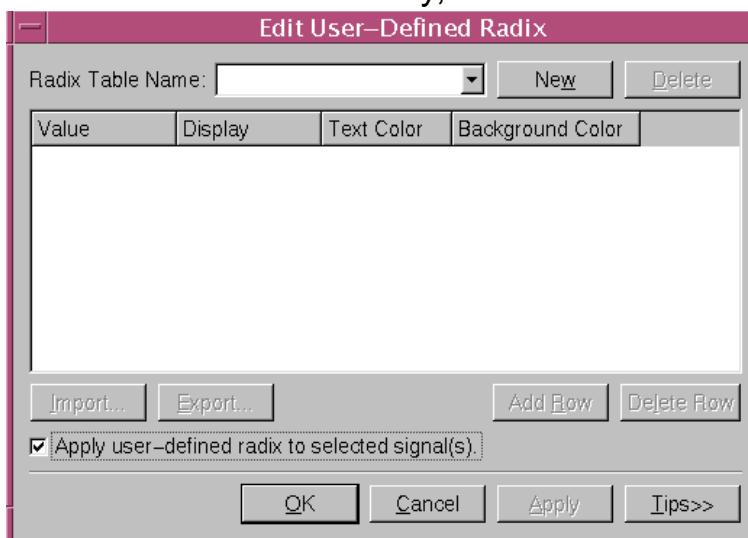
If you do not toggle Default Properties off, the changes will become the default and duplicate signal display will also change.

If a signal group is in two Wave views, changing a signal will change the signal in the other Wave view if it is the same instance.

Creating a User-defined Radix

You can define a custom mnemonic mapping from values to strings for display in the Wave view.

1. Select **Signal > Set Radix > User Defined > Edit** to display the Edit User-defined Radix dialog box.
2. To create a user-defined radix, click **New**, enter a radix name, then press Return.
3. Click Add Row to activate a row for the user-defined radix. For each row entry, select the text and background colors.



4. Click **OK** or **Apply** to save the user-defined radix.
5. To apply the user-defined radix to a signal, select the signal in the Wave view, select **Signal > Set Radix**, then select the user-defined radix from the list.

Managing User-defined Radices

To edit or delete a user-defined radix:

1. Select **Signal > Set Radix > User Defined > Edit** to display the Edit User-defined Radix dialog box.
2. To delete a user-defined radix, select the radix from the User-defined Radix pull-down menu, then click **Delete**.

To edit a user-defined radix, select the radix from the User-defined Radix pull-down menu, click a cell in the Value or Display table, then enter your change.

3. Click **OK** or **Apply** to save the change.

Importing and Exporting a User-defined Radix

To import or export a user-defined radix:

1. Select **Signal > Set Radix > User Defined > Edit** to display the Edit User-defined Radix dialog box.
2. To import a radix click **Import**, then browse to and select the desired radix.

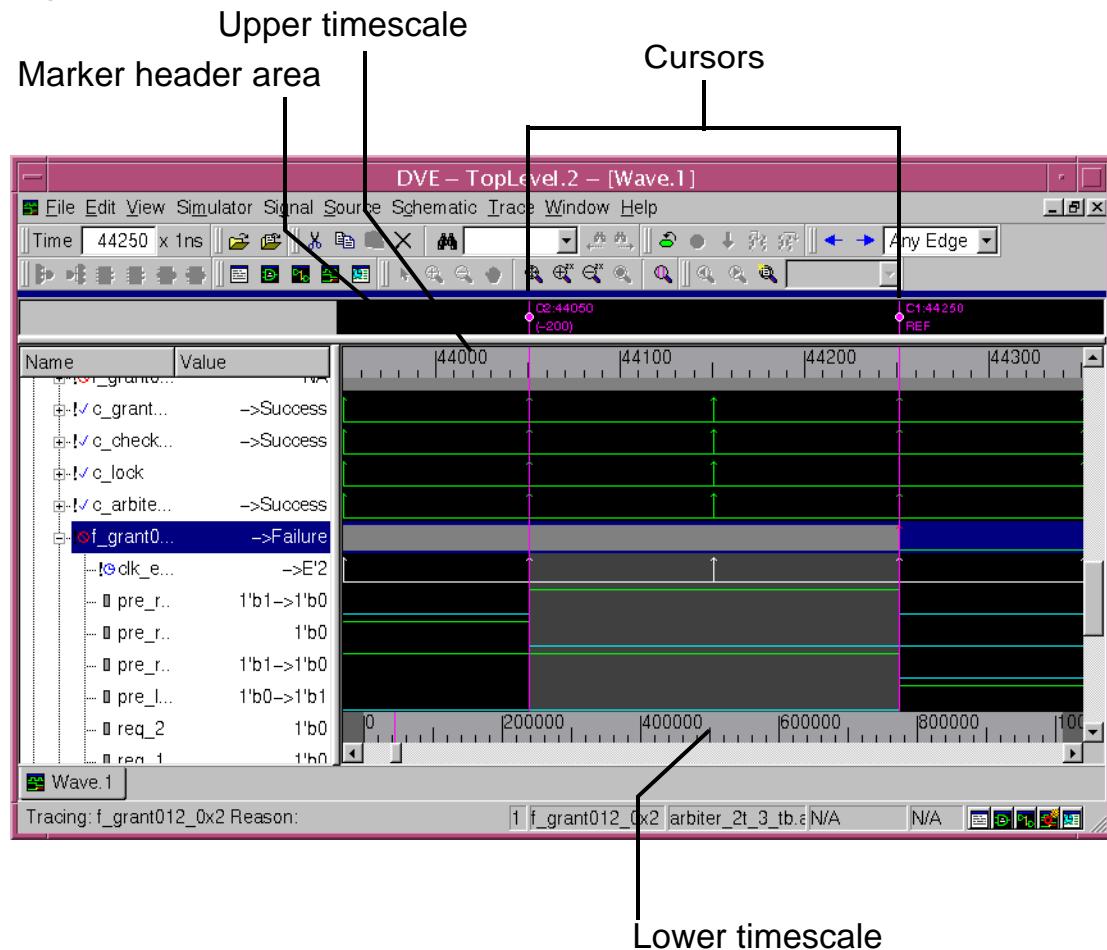
To export a user-defined radix, click **Export**, select the radix from the User-defined Radix pull-down menu, then enter a radix name.

3. Click **OK** or **Apply**.

Using the Waveform Pane

The Wave view displays the value transitions of signals and the success or failure of assertions.

Figure 8-13 The Wave view



Cursors and markers are explained in “[Using Cursors](#)” on page 31.

The Wave view has an upper and a lower timescale. The upper timescale displays the range of simulation times currently on display in the Wave view. The lower timescale displays the range of simulation times throughout the entire simulation.

Using Cursors

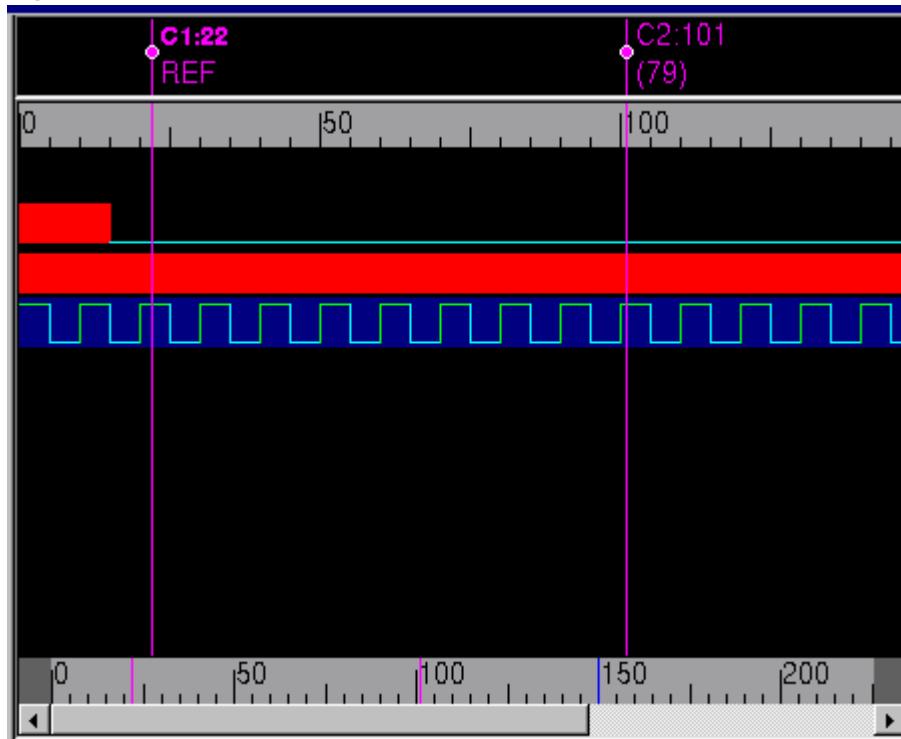
To insert cursors:

- Click the left mouse button to deposit cursor C1 in the graphical display. The C1 cursors default position is at time 0. Left click somewhere else in the graphical display and cursor C1 moves to this new location.
- Click the middle mouse button to deposit cursor C2 in the graphical display. Similar to cursor C1, middle click somewhere else in the graphical display and cursor C2 moves to this new location.

To move cursors, place the mouse cursor on the round cursor handle in the cursor area, hold down the left mouse button and drag the cursor to the desired location. You can click either the left or the middle mouse button in the waveform or cursor area to move C1 or C2 respectively.

The interval between the two cursors is always displayed in the marker header area.

Figure 8-14 Graphical Display Cursors



As shown in [Figure 8-14](#), the simulation time and the delta between the reference cursor (C1) and cursor C2 is shown in the marker header area.

Using Markers

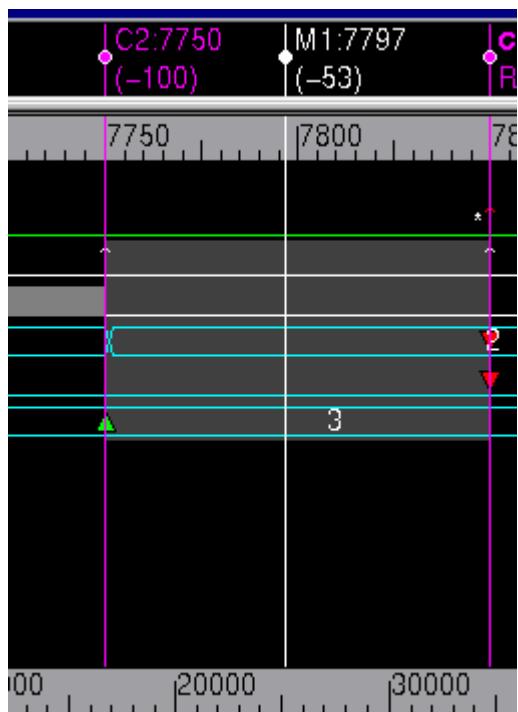
Markers differ from cursors in the way you insert and move them. Like cursors, markers display the delta between the reference cursor (C1) and the marker.

To insert a marker you can use the Markers dialog box, see [“Moving, Hiding, and Deleting Markers” on page 34](#), or perform the following:

1. Right click in the graphical display. This displays the context-sensitive menu (CSM) for the graphical display.

2. Select **Create Marker** from this menu. This inserts a dotted line on your mouse cursor in the graphical display.
3. The dotted line tracks the mouse cursor as you move the mouse in the waveform or marker header area. Position the marker in the graphical display then left click to position the marker. Note that the marker annotation displays marker position and the delta between the marker and cursor C1.

Figure 8-15 New Marker

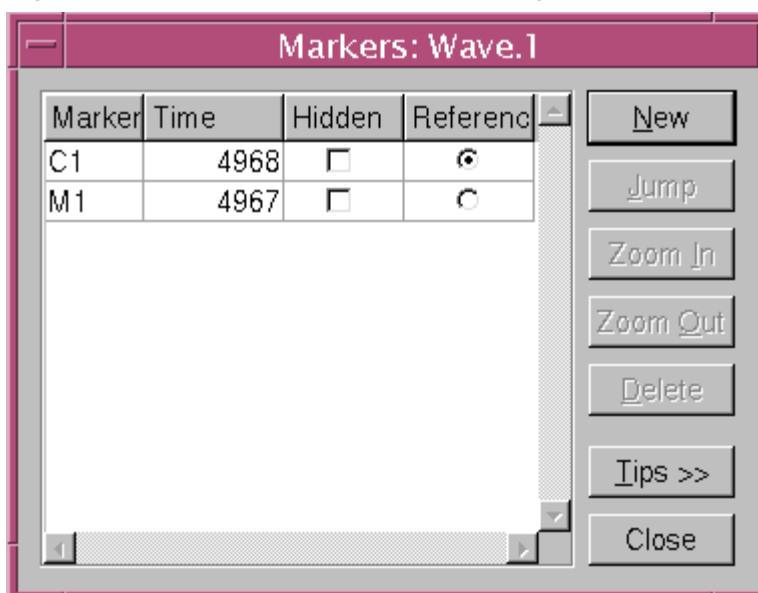


As you insert markers, DVE names them M1, M2, M3 and so forth. You can rename them using the Markers dialog box.

Moving, Hiding, and Deleting Markers

The Markers dialog box allows you to create, move, hide, and delete markers, set the reference marker, and to scroll the graphical display until it reveals a marker. To open this dialog box, right-click in the Waveform pane, and select Markers from the context-sensitive menu.

Figure 8-16 The Markers Dialog Box



The C1 marker is the default reference marker. To set another marker as the reference marker, select the marker in the reference column.

Click the **Tips** button to expand the dialog box to show context-sensitive help for the dialog box.

Searching in the Graphical Panes

When searching graphical panes, if any signals are selected in the wave view, searching will search only in the waveforms for the selected signals. If no signal is selected, it searches all the signals.

You can have the C1 cursor move from its current location to the next

using the search forward and search backward arrows  in the toolbar. To set the search criteria, click the down arrow in the toolbar and select one of the following:

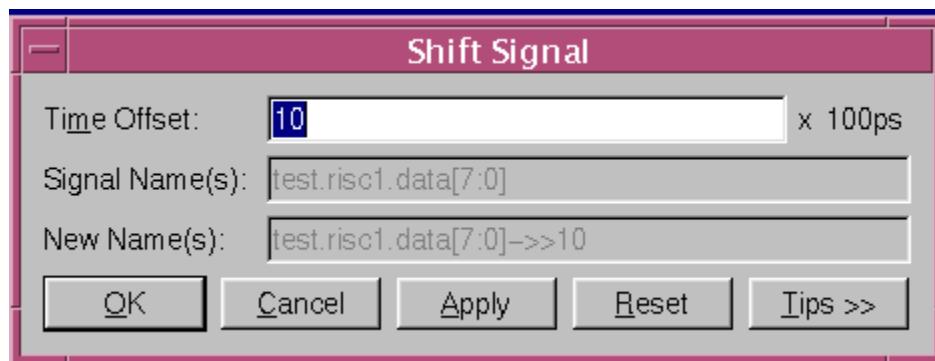
- Any Edge
- Rising
- Falling
- Match
- Mismatch
- Value

Shifting Signals

You shift a signal by creating a new signal based on a time shifted signal.

1. Select a signal in the Signal pane.
2. Select **Signal > Shift Time** to display the Shift Signal dialog box.

3. Enter a positive Time Offset to shift the signal to the right or a negative number to shift to the left in the Waveform pane.

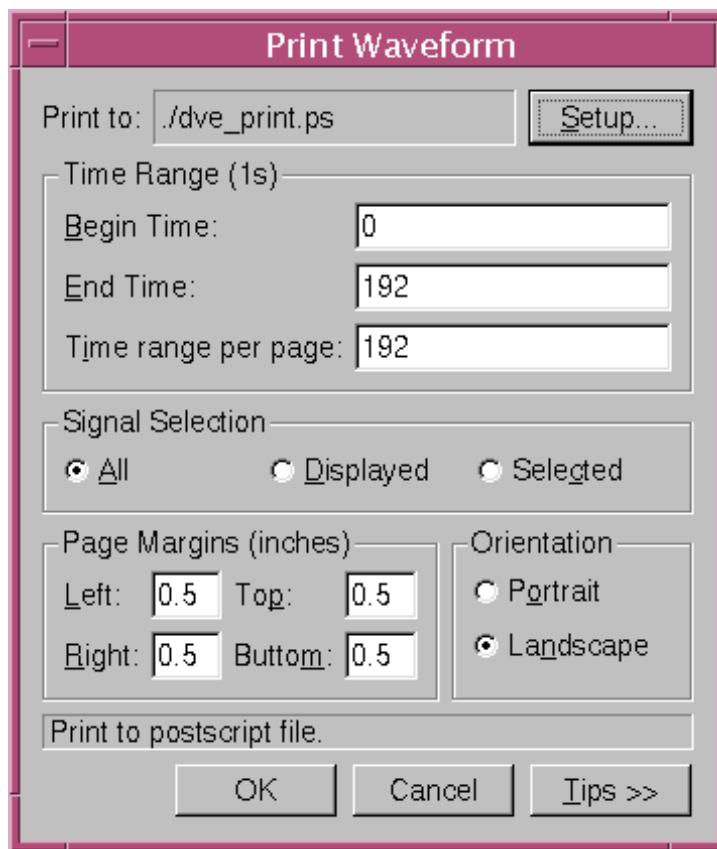


The signal displays with the original signal name followed by the time offset. In this illustration this information is shown as:
test1.risc.daata (7:0) ->>10.

Printing Waveforms

You can print waveforms to a file or printer from an active Wave view selecting time range and signals to print.

1. From an active Wave view, select **File > Print** or click the print toolbar button  to display the Print Waveform dialog box.



2. Click **Tips** for printing information.
3. Click **OK** to print.

Using Schematics

Schematic views provide a compact, easy-to-read graphical representation of a design. View a design, scope, signal or group of selected signals and select ports to expand connectivity in relevant areas. Explore the design behavior by analyzing the annotated values for ports and nets.

There are two types of schematic views in DVE: design and path.

- A design schematic shows the hierarchical contents of a design or a selected instance and lets you traverse the hierarchy of the design.
- A path schematic is a subset of the design schematic displaying where signals cross hierarchy levels. Use the path schematic to follow a signal through the hierarchy and display portal logic (signal effects at ports).

Note:

Your design must be compiled in the same version of VCS that you are currently pointing to in your session.

Opening a Schematic View

To view a schematic of an open design in DVE, you must generate your simulation on the same platform you run DVE from and you must use the `-debug` command-line option to compile.

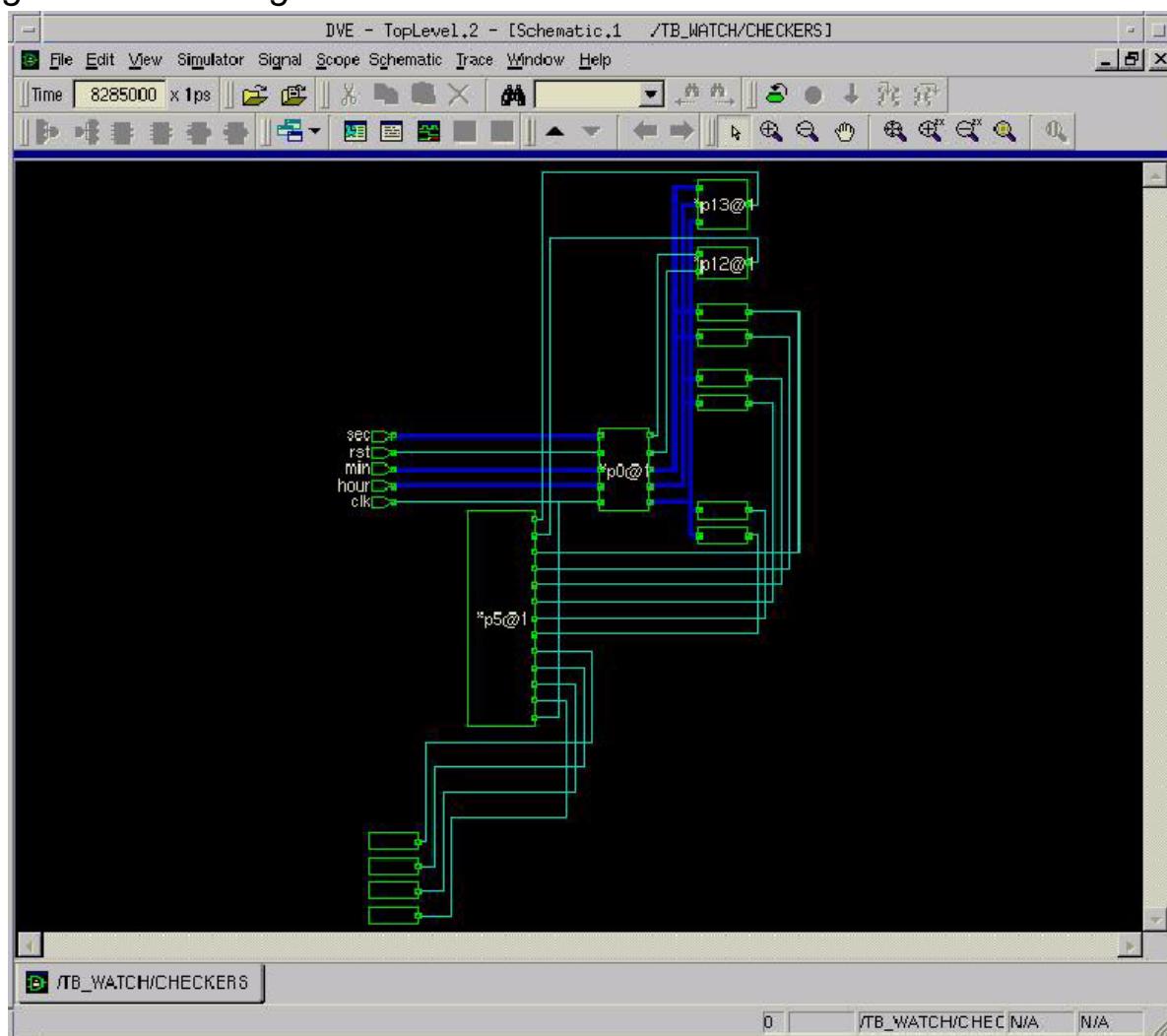
To open a display design schematic perform the following:

1. To choose between opening the schematic in the current active DVE window or in a new window, click  in the status bar to toggle a new window on or  to view the schematic in the current window.

2. Select an instance from the Hierarchy Browser, right-click, then select **Show Schematic** or **Show Path Schematic** from the context-sensitive menu.

The Schematic view displays the connectivity in the selected instance. [Figure 8-17](#) shows an example of a hierarchical schematic.

Figure 8-17 Design Schematic



Selecting a Signal in the Schematic

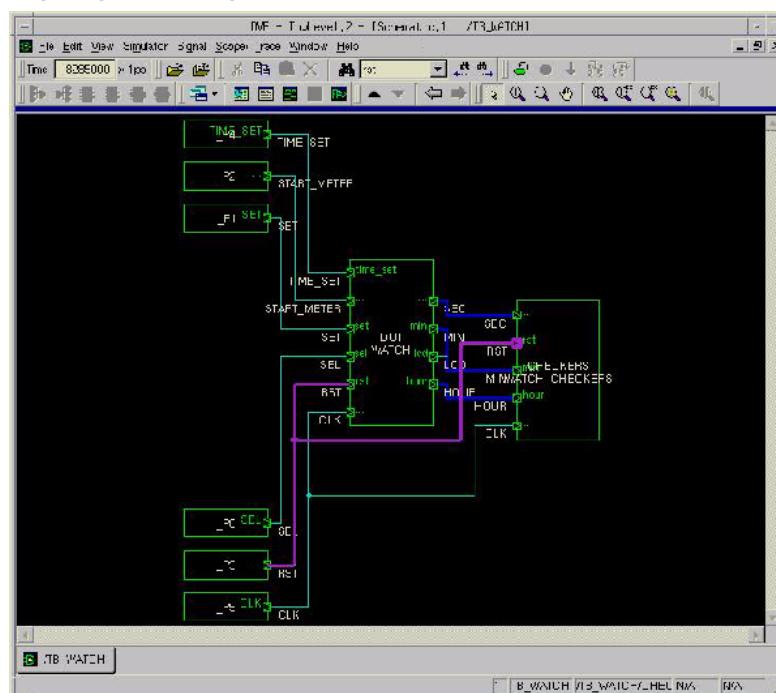
1. Click on a signal in the schematic or, to find the signal in the schematic, enter the signal name in the Find toolbar box in the Schematic view, then click the Find Next toolbar button.

The signal becomes highlighted in the schematic.

2. With the signal selected, click the Trace Drivers toolbar button, or select the **Trace-Drivers** menu item.

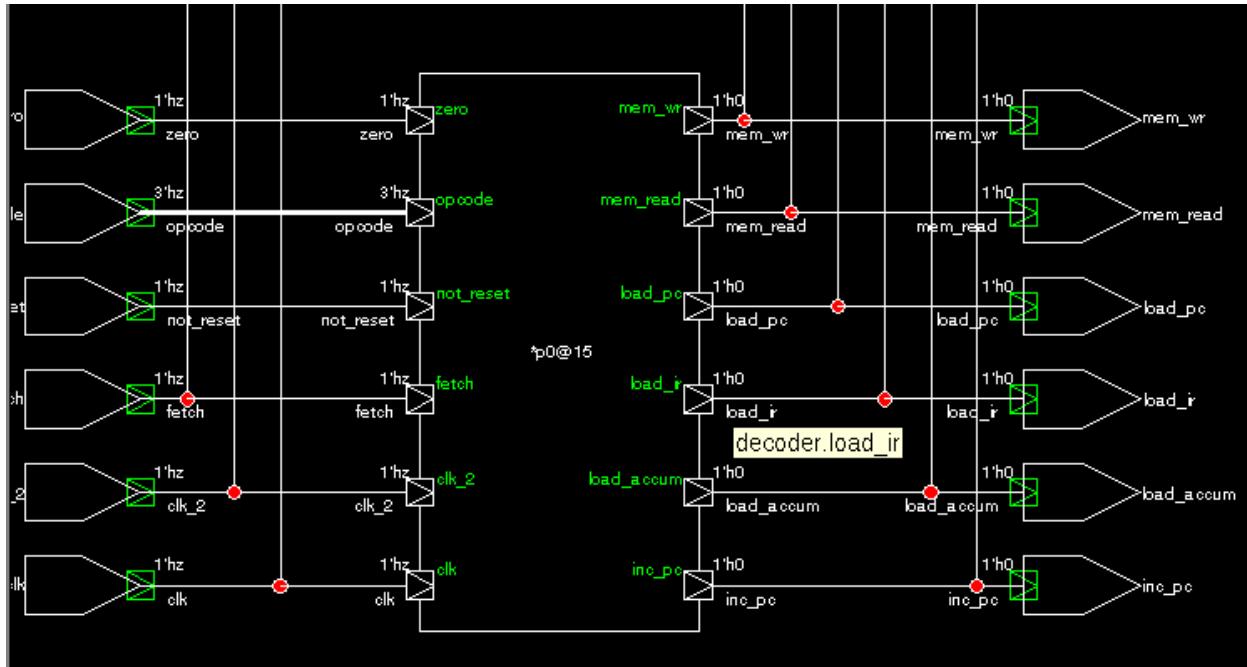
The signal is now highlighted in purple.

Figure 8-18 *Highlighted signal in Schematic view*



Annotating Values

Select Schematic > Annotate Values.



Note:

If you hold the cursor on a signal, a ToolTip identifies the signal as shown above.

Displaying Connections in a Path Schematic

With a path schematic displayed, you can add the logic fanin to, or logic fanout from, specified objects in the schematic across specified levels or the entire design. To display connections:

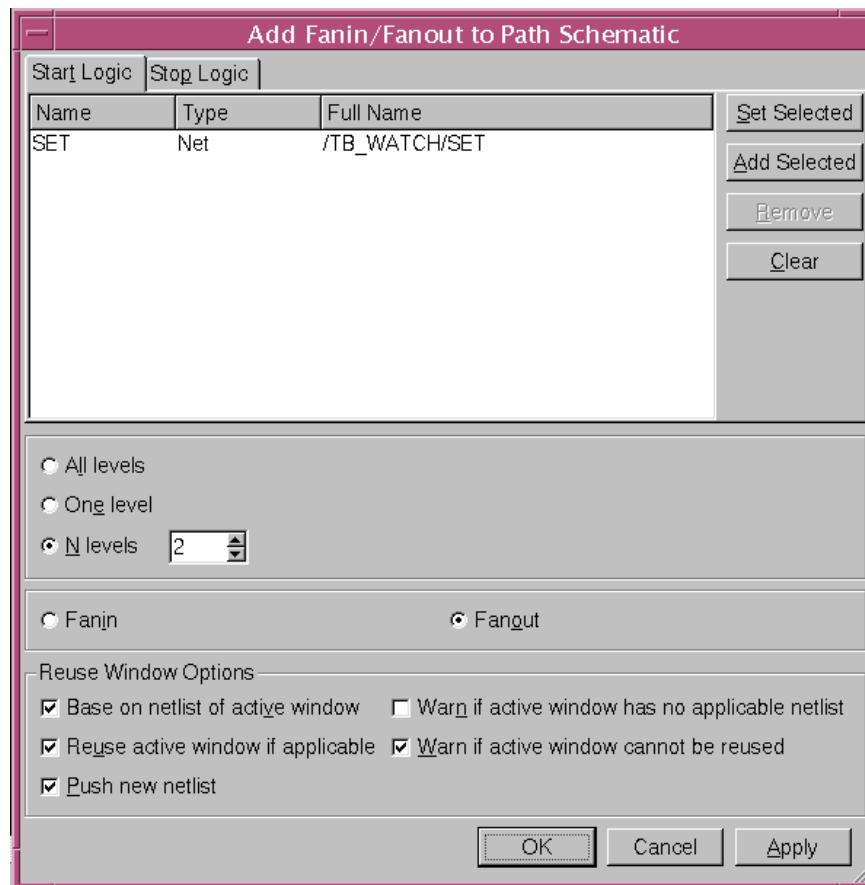
1. Select an object or objects in the path schematic.

Your selection changes color confirming selection.

2. Select **Scope > Add Fanin/Fanout**.

The Fanin/Fanout to Path Schematic dialog box displays as shown in [Figure 8-19](#).

Figure 8-19 Add Fanin/Fanout to Path Schematic dialog box

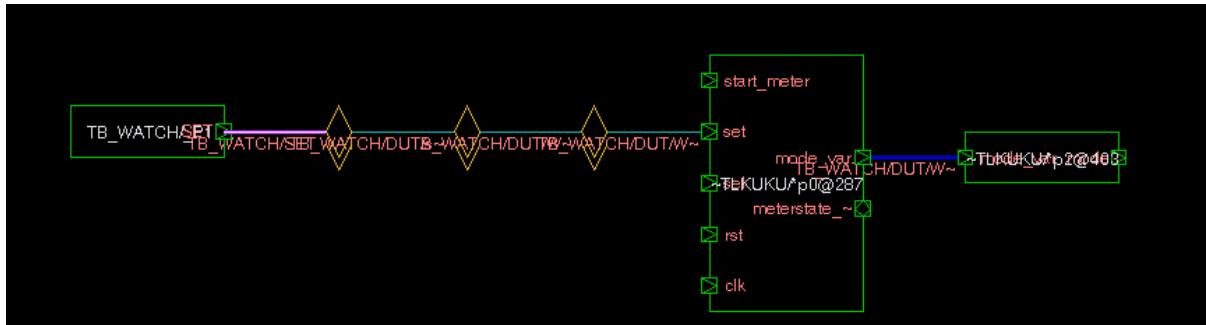


3. Click **Set Selected** to add the selected objects to the list box. You can optionally select more objects and use the **Add Selected** button to add them to the list.
4. Set the other options such as the number of logic levels to be added and the Reuse views Options.

- Click **OK** to update the schematic with the additional fanin or fanout logic.

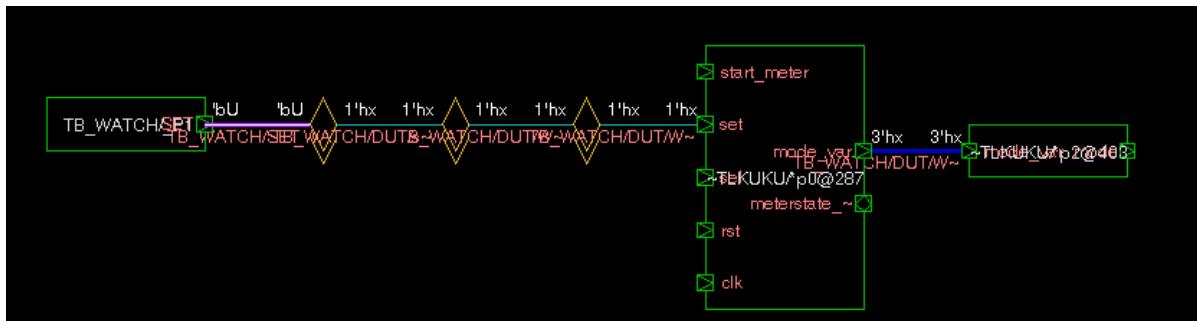
The schematic displays as shown in [Figure 8-20](#).

Figure 8-20 Path schematic with fanin logic



- Select **Signal > Annotate Values** to view signal values.

Figure 8-21 Path Schematic with values



Manually Tracing Signals

You can select one or more signals to trace in a Schematic or Path Schematic view. With this option, the selected signals are highlighted based on specified line colors you select.

To highlight signals:

- Select **Trace > Highlight**, then select **Set Current Color**.

2. Select a color for the highlight.
3. Click **OK**.

Searching for Signals

You can use the Find toolbar option to locate signals.

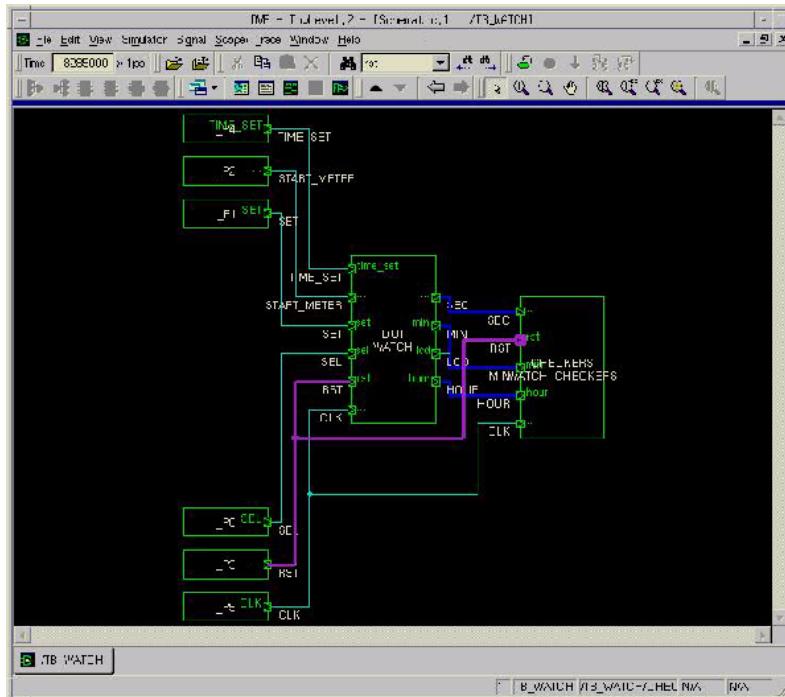
1. Click on a signal in the schematic or, to find the signal in the schematic, enter the signal name in the Find toolbar box in the Schematic view, then click the Find Next toolbar button.

The signal becomes highlighted in the schematic.

2. With the signal selected, click the Trace Drivers toolbar button, or select the **Trace-Drivers** menu item.

The signal is now highlighted in purple.

Figure 8-22 Highlighted signal in Schematic view

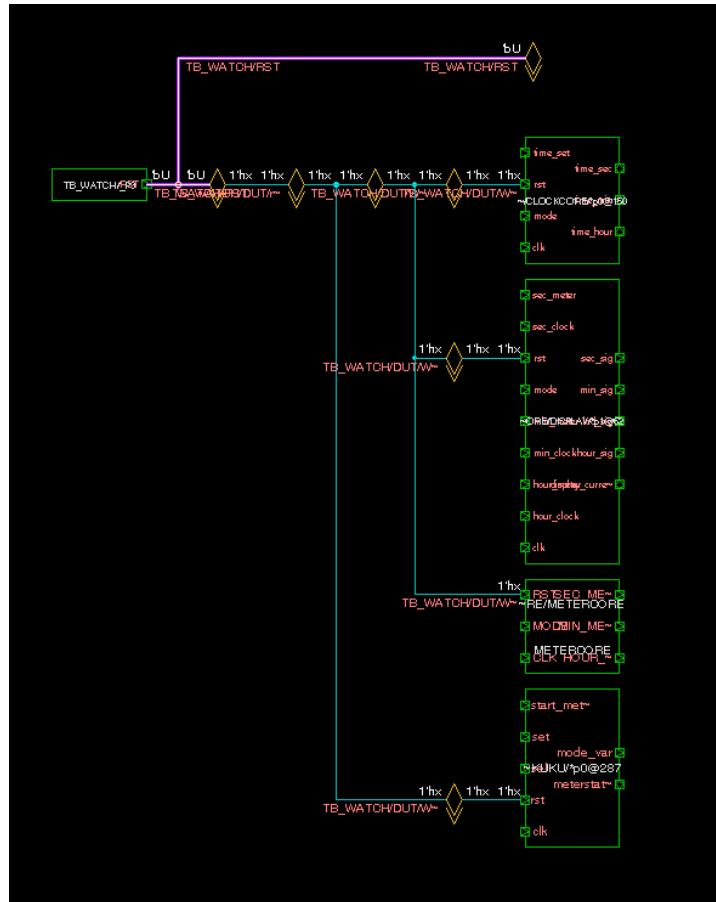


Following a Signal Across Boundaries

You can select a signal and follow it across hierarchical boundaries in the Path Schematic view.

1. Select a signal or signals in any DVE list, then right-click and select **Show Path Schematic**.
2. In the Path Schematic, select a signal.
3. Right-click and select **Expand Path** from the context-sensitive menu.

The signal is highlighted in the path view.



Tracing X Values in a Design

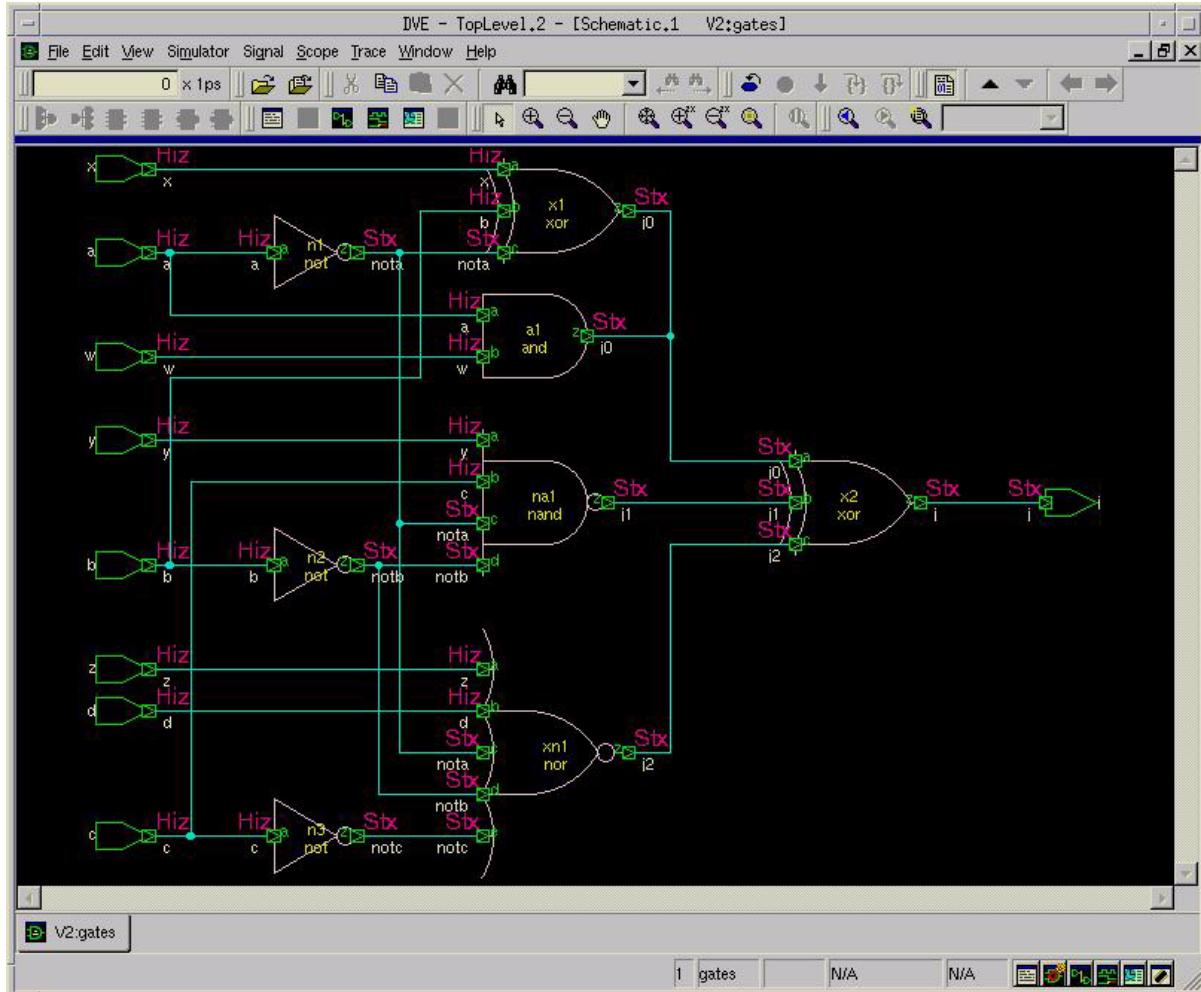
You can trace an X value through a design, for example, across gates, to identify the signal that caused the X value.

To trace an X value:

1. Select an instance from the Hierarchy Browser, then right-click and select **Show Schematic**.

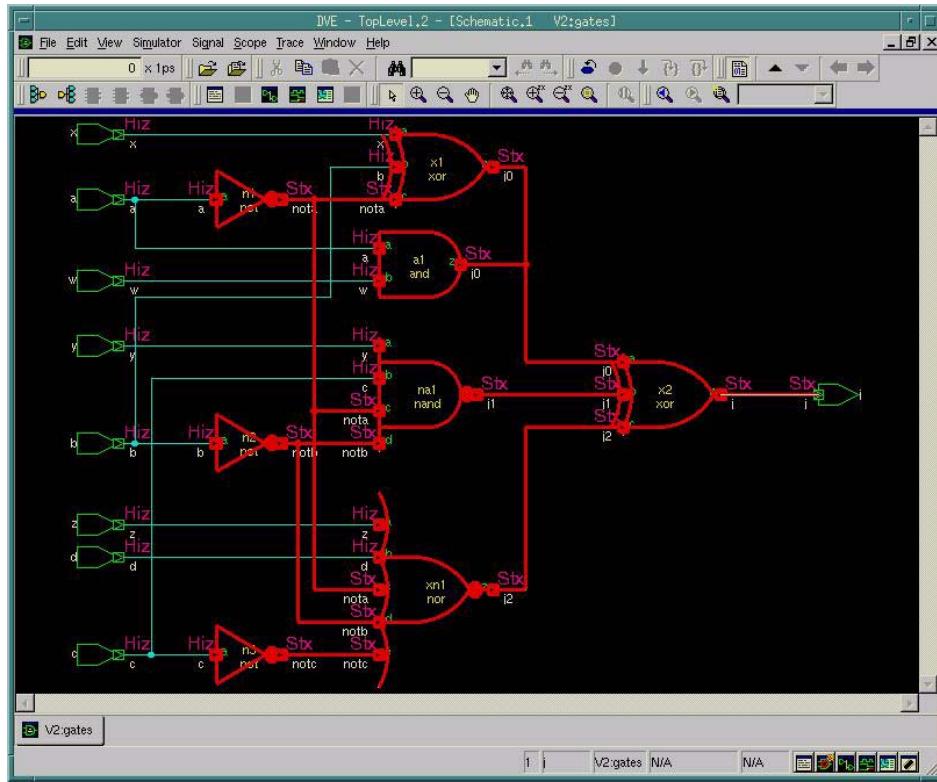
By default, a new Schematic view opens in the main view as a tabbed view.

2. Zoom schematic to fit view.
3. Click the **Annotate Values** toolbar button in the Schematic view.



4. Select Schematic view category.
5. Select the signal with an X value. The signal wire turns white when selected.
6. Select **Trace > Trace X**.

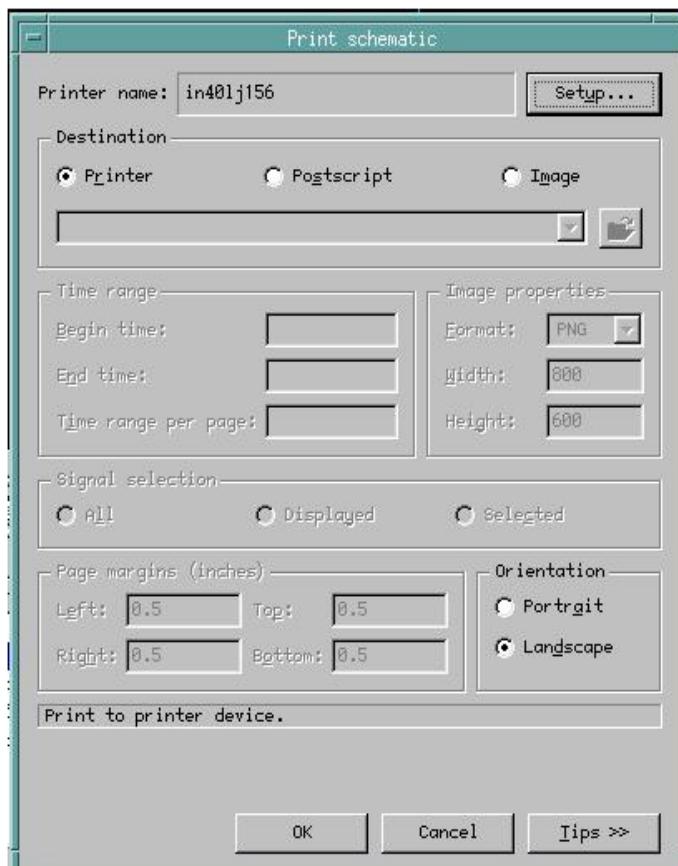
The x value is traced to its source signals making it easy to identify the signals that caused the X value.



Printing Schematics

You can print schematics to a file or printer from an active Schematic or Path Schematic view selecting time range and signals to print.

1. From an active Schematic or Path Schematic view, select **File > Print** or click the print toolbar button  to display the Print Schematic dialog box.



2. Click the Setup button to set printing options:
 - Printer or print to file
 - Print in color or grayscale
 - Print orientation and paper size
 - Print options such as range and number of copies
3. Select whether to print **All**, **Displayed**, or **Selected** signals.
4. Select the page margins.

5. Choose **Landscape** or **Portrait** orientation.
 6. Click **OK** to print.
-

Working with Assertions and Cover Properties

Use the `-assert dve` flag on the VCS command line when compiling SystemVerilog assertions (SVA) for debugging with DVE.

Tip: The link step can take a long time if you use a Solaris linker prior to version 5.8. To avoid linking delays when using DVE to debug designs compiled on Solaris, do either of the following:

- Make sure your Solaris C compiler is version 5.8 or above. To check your compiler version, enter the following on the command line:

```
% ld -v
```

The system returns your linker version, for example:

```
ld: Software Generation Utilities - Solaris Link Editors:  
5.8-1.283
```

- Use the gcc C compiler when compiling your design. For example:

```
% vcs -assert dve -PP -sverilog top.V -cc gcc
```

Viewing Assertion Results

The Assertion Pane displays SVA and OVA assertion and cover property results.

- DVE displays assertion results by instance include start and end times of assertion events, the delta, the offending string in assertion failures, and totals for failures, successes, incompletes and attempts. [Figure 8-23](#) shows the Assertion pane with successful assertions displayed in green and failed assertions in red.

Note:

Use the VCS -assert dve command-line switch with the -PP flag to enable SVA tracing in DVE. If you do not enable SVA tracing, assertion value changes will still be dumped into the VPD file and be visible in the Wave view, but assertion attempts cannot be traced.

- Cover properties results, shown in [Figure 8-23](#), display instance include start, end time, and the delta, and the total number of matches, mismatches, incompletes, and attempts.

Setting Display Criteria

You use the Assertion Pane navigation bar to display results based on selected criteria as shown in [Figure 8-23](#).

Figure 8-23 Assertion and cover properties displayed

The screenshot shows two tables side-by-side. The top table is titled "Assertions" and the bottom table is titled "Cover properties". Both tables have columns for Name, Instance, Start, End, Delta, Reason, Failures, Incompletes, Successes, and Attempts. Below each table is a display control bar with filters for Name and Instance, and dropdowns for time (at time, current, from, to, end), and condition (failures, incompletes, successes, all). The "Cover properties" table also has a column for Matches and Mismatches.

Annotations:

- Click to display assertion or cover properties results.** Points to the "Assertions" table title.
- Specify display criteria by starting and ending time.** Points to the "at time" dropdown in the "Assertions" control bar.
- Select a condition for the results display.** Points to the "all" radio button in the "Cover properties" control bar.

Name	Instance	Start	End	Delta	Reason	Failures	Incompletes	Successes	Attempts
HOUR_24_CHECKER /TB_WATCH/CHECKERS0		0	0			0	0	10000	10000
MIN_60_CHECKER /TB_WATCH/CHECKERS0		0	0			0	0	10000	10000
RESET_CHECKER /TB_WATCH/CHECKERS0		8295000	0		((hour == 5'b0) && (min == 15))	0	19985	20000	
SEC_60_CHECKER /TB_WATCH/CHECKERS0		9410000	10000	(sec == 6'b0)		137	0	9863	10000

Name	Instance	Start	End	Delta	Matches	Mismatches	Incompletes	Attempts
HOUR_24_COVER /TB_WATCH/CHECKERS0		0	0		10000	0	0	10000
MIN_60_COVER /TB_WATCH/CHECKERS0		0	0		10000	0	0	10000
RESET_COVER /TB_WATCH/CHECKERS0		0	0		19985	0	0	19985
SEC_60_COVER /TB_WATCH/CHECKERS0		0	0		9863	0	0	9863

To set display criteria by starting ending time, perform the following:

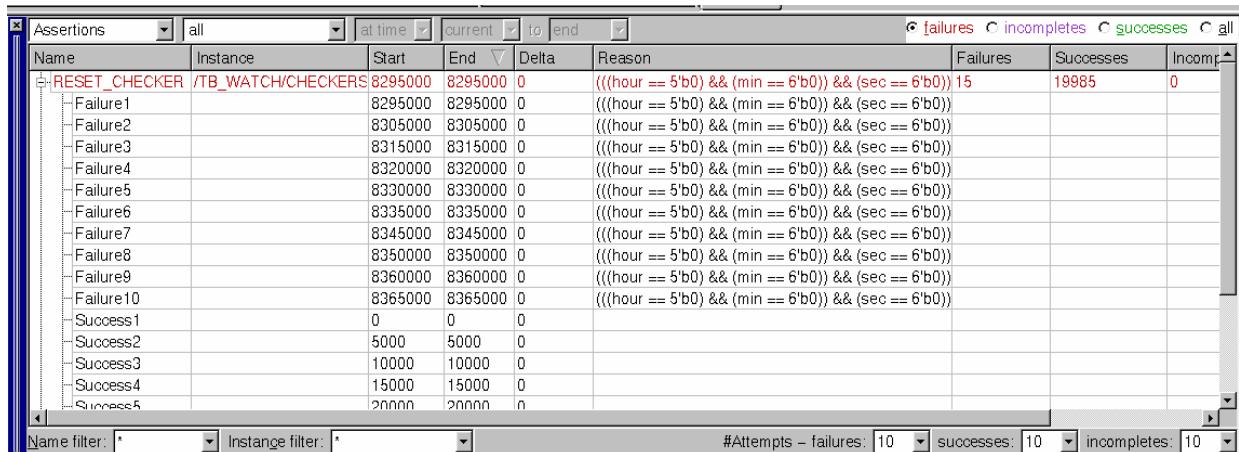
1. Click the arrow in the display control bar and select **all**, **starting**, **ending**, or **starting and ending**.
2. Select **at time** or **from**, then select **begin**, **current**, or **end** to specify the view.
3. Specify the condition as follows:

- For assertions, select failures, incompletes, successes, or all.
- For cover properties select **uncovered**, **covered** or **all**. Note that the default is to display uncovered.

Debugging Assertions

When you open a design that contains assertions, DVE displays the Assertion Pane even if all the assertions pass. The default is to display failed assertions. To display the first 10 failures and successes, click the + next to an assertion of interest. [Figure 8-24](#) shows the attempt list.

Figure 8-24 Assertion attempt list



The screenshot shows the DVE Assertion pane with the following details:

Name	Instance	Start	End	Delta	Reason	Failures	Successes	Incompl.
RESET_CHECKER	/TB_WATCH/CHECKERS	8295000	8295000	0	<code>((hour == 5'b0) && (min == 6'b0)) && (sec == 6'b0))</code>	19985	0	
Failure1		8295000	8295000	0	<code>((hour == 5'b0) && (min == 6'b0)) && (sec == 6'b0))</code>			
Failure2		8305000	8305000	0	<code>((hour == 5'b0) && (min == 6'b0)) && (sec == 6'b0))</code>			
Failure3		8315000	8315000	0	<code>((hour == 5'b0) && (min == 6'b0)) && (sec == 6'b0))</code>			
Failure4		8320000	8320000	0	<code>((hour == 5'b0) && (min == 6'b0)) && (sec == 6'b0))</code>			
Failure5		8330000	8330000	0	<code>((hour == 5'b0) && (min == 6'b0)) && (sec == 6'b0))</code>			
Failure6		8335000	8335000	0	<code>((hour == 5'b0) && (min == 6'b0)) && (sec == 6'b0))</code>			
Failure7		8345000	8345000	0	<code>((hour == 5'b0) && (min == 6'b0)) && (sec == 6'b0))</code>			
Failure8		8350000	8350000	0	<code>((hour == 5'b0) && (min == 6'b0)) && (sec == 6'b0))</code>			
Failure9		8360000	8360000	0	<code>((hour == 5'b0) && (min == 6'b0)) && (sec == 6'b0))</code>			
Failure10		8365000	8365000	0	<code>((hour == 5'b0) && (min == 6'b0)) && (sec == 6'b0))</code>			
Success1		0	0	0				
Success2		5000	5000	0				
Success3		10000	10000	0				
Success4		15000	15000	0				
Success5		20000	20000	0				

At the bottom of the pane, there are filters for Name filter, Instance filter, #Attempts – failures: 10, successes: 10, and incompletes: 10.

The Assertion Pane is interconnected with other DVE windows and panes. To display an assertion in the Hierarchy and Variable panes, the Source view and the Wave view, double click an assertion instance in the Assertion Pane.

The following occurs:

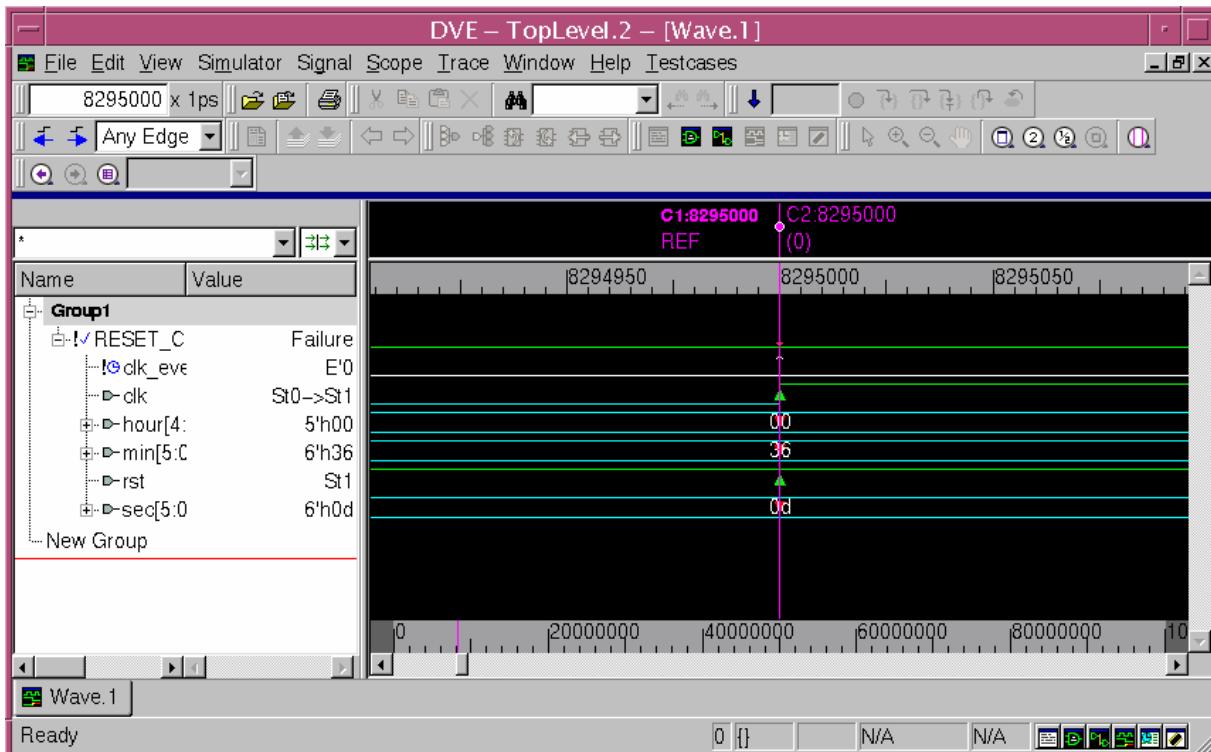
- The Hierarchy Pane displays the associated unit or HDL scope that contains the assertion.

- The Variable Pane displays the HDL variables corresponding to the unit or scope. This is not specific to the assertion (i.e., it may contain more signals that are used in the assertion).
- Up to three source views may be displayed: one for the HDL source, one for the bind source and one for the OVA definition source.
- The Wave view opens and displays the selected assertion centered on the failure. The cursors mark the start and end time of the selected assertion with the area between the cursors grayed. A green circle indicates a signal value at a specific time that contributed to successful sub-expression in the assertion. A red circle indicates a signal value at the time which caused a sub-expression to fail. A failing sub-expression may result in the overall assertion failing.

Viewing an Assertion in the Wave view

[Figure 8-25](#) shows an assertion with no delta between start and end time.

Figure 8-25 Assertion in Wave view



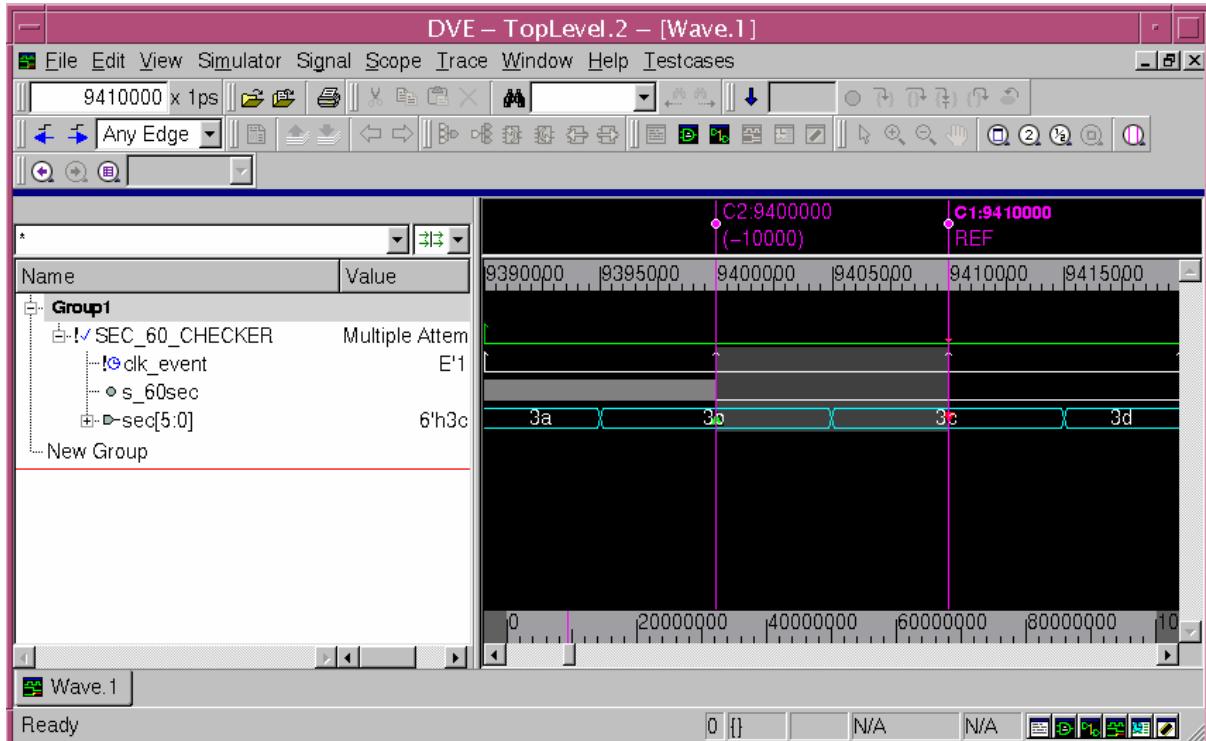
- The C1 and C2 cursors are automatically placed at the start and end of the assertion. In this assertion, we did not have a time component (sequence), so C1 and C2 are at the same time.
- In Signal Group 1, the assertion RESET_CHECKER is listed first in the tree view. This is the assertion result signal. The waveform consists of red and green arrows. Green shows where the assertion was determined to be a success, red shows where it failed. The red arrow indicates the first failure.
- RESET_CHECKER is expanded into components.
 - The first component is clk_event. Each clock event shows you when the assertion fired and the clock ticks that happen for sequences.

- The rest of the signals are the signals that contributed to the success or failure. The signals are clk, rst, hour, min, sec.
- The green dots on the waveform show you that the signal was OK at that clock tick. The red dots show you that the signal contributed to the failure of the assertion at that clock tick.

Viewing an Assertion Failure Time Delta

When there is a delta for an assertion failure, the Wave view opens and displays the selected assertions centered on the failure. The cursors mark the start and end time of the selected assertion with the area between the cursors grayed. A green circle within a signal indicates a successful signal or value, while a red inverted circle and the C1 cursor indicates a failure.

Figure 8-26 Assertion failure time delta in the Wave view



- If you hold the mouse cursor over a green or red circle, an InfoTip pops up and shows details on the impact of the signal.
 - If a green arrow, the InfoTip tells why this signal contributed to a success so far.
 - If a red arrow, the InfoTip tells why this signal contributed to a failure.
- A white arrows indicate assertion clock events in the view.
- If you hold the mouse cursor over the attempt failure, an InfoTip pops up and shows details on the failure. The InfoTip contains (for each attempt ending at this time):
 - Start time
 - End time

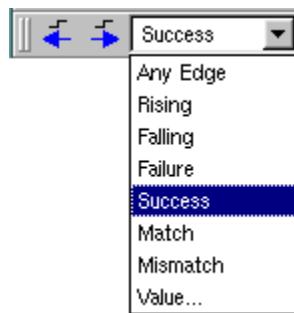
- Delta
- Instance
- Offending/Reason

Viewing Successes and Failures

Use the Search toolbar item to move to the next or previous assertion success or failure in the Wave view.

1. With an assertion selected, select **Success** or **Failure** from the Search pull-down menu .

Figure 8-27 Navigate assertion successes and failures



2. Click the back or forward arrow to move to the previous or next success or failure of the selected assertion.

Note:

The Match and Mismatch items in the Search pull-down menu are used for signal comparisons, not for assertion cover property match and mismatch display in the Wave view.

Local variables are inserted into the signal list in the hierarchy pane as shown by the local variable count..



Viewing Source Code

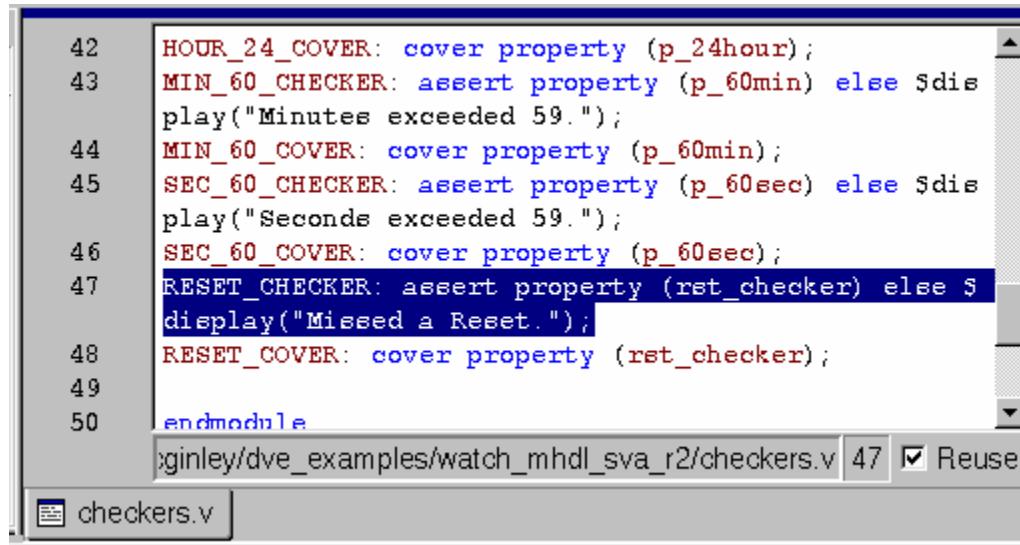
To display code related to an assertion attempt:

1. Double-click an assertion attempt in either Assertion view tab.

Code is displayed as follows:

- The Source Pane displays the HDL code where the assertion is inlined or bound.
- A Source view displays the assertion code with the assertion highlighted (see [Figure 8-28](#)).

Figure 8-28 Assertion source code



The screenshot shows a source code editor window with the following content:

```
42 HOUR_24_COVER: cover property (p_24hour);
43 MIN_60_CHECKER: assert property (p_60min) else $display("Minutes exceeded 59.");
44 MIN_60_COVER: cover property (p_60min);
45 SEC_60_CHECKER: assert property (p_60sec) else $display("Seconds exceeded 59.");
46 SEC_60_COVER: cover property (p_60sec);
47 RESET_CHECKER: assert property (rst_checker) else $display("Missed a Reset.");
48 RESET_COVER: cover property (rst_checker);
49
50 endmodule
```

The line `RESET_CHECKER: assert property (rst_checker) else $display("Missed a Reset.");` is highlighted in blue, indicating it is the selected assertion. The status bar at the bottom right shows the file name `checkers.v`, line number `47`, and a checked checkbox labeled `Reuse`.

2. To edit the assertion in your default text editor, select **Edit > Edit Source**.

9

Using Unified Command-line Interface

UCLI provides a common set of commands for Synopsys verification products. UCLI commands are built based on Tcl. Therefore, you can use any Tcl command with UCLI. You can also write Tcl procedures and execute them at the UCLI prompt.

This chapter describes the following with the usage model and examples:

- “[Invoking UCLI](#)”
- “[Commonly Used UCLI Commands](#)”

Invoking UCLI

The usage model to invoke UCLI is as shown below:

Compilation

```
% vcs [vcs_options] -debug_pp|-debug|-debug_all \
        file1.v file2.v file3.v
```

Simulation

```
% simv [simv_options] -ucli
```

To invoke UCLI, ensure that you specify the `-debug_pp`, `-debug`, or `-debug_all` options during compilation. You can then use the `-ucli` option at runtime to enter the UCLI prompt at time 0 as shown in the following example:

```
% simv -ucli
Chronologic VCS simulator copyright 1991-2005
Contains Synopsys proprietary information.
Compiler version Y-2006.06-SP1-6; Runtime version Y-2006.06-
SP1-6; Sep 4 12:31 2007

ucli%
```

UCLI Related Compile-time and Runtime Options

The following section describes the UCLI related compile-time options:

`-debug_pp`

Enables you to control the simulation, save-restore simulation state, get driver and load information, and run Tcl macro routines.

`-debug`

In addition to all that `-debug_pp` offers, `-debug` allows you to force/release a reg/wire/signal.

`-debug_all`

In addition to all that `-debug_pp` and `-debug` offer, `-debug_all` allows you to set breakpoints and perform line stepping.

The following are the UCLI-related runtime options:

`-ucli`

Invokes UCLI prompt.

`-do command_file`

Invokes UCLI, and executes the commands specified in the *command_file*.

Commonly Used UCLI Commands

You can execute the following tasks in UCLI:

- “Using the help Command”
- “Tool Environment Array Commands”
- “Controlling the Simulation”
- “Dumping a VPD File”
- “Setting Breakpoints”
- “Forcing and Releasing Signals”
- “Calling System Task or Function”
- “Macro Control Routine”
- “Navigation Command”
- “Design Query Commands”

The following section describes the basic usage of the UCLI commands to accomplish the above mentioned tasks. For further information on UCLI commands, use the `help` command or refer to the *Unified Command-line Interface User Guide*.

Using the help Command

`help UCLI_command`

You can use the `help` command to:

- List all UCLI commands.

- See the description and the usage model of the specified UCLI command.

Example

The following command shows the usage of the help command:

```
ucli% help senv
senv      Display one or all synopsys::env array elements
usage:
    senv [name]
        where 'name' is a synopsys environment entry
```

Tool Environment Array Commands

senv [element]

senv displays the environment array elements such as current time, time precision, active scope and so on.

Example

```
ucli% senv time
140 ns
ucli%
```

Controlling the Simulation

run [-relative | -absolute] [time]

You can use the **run** command to:

- Run the complete simulation.
- Run until the specified relative or an absolute time. By default, **run time** considers the specified time as the relative time.

Example

The following command runs the simulation until the end of simulation or until a breakpoint (if set) is encountered.

```
% simv -ucli  
ucli% run
```

The following command, runs the simulation until 1000 ns. If the time unit is not specified, VCS obtains the timescale based on the timescale set during compilation.

```
% simv -ucli  
ucli% run 1000 ns
```

next

You can use the next command to advance the simulation stepping over tasks and functions.

Note:

You should compile the design using `-debug_all` to use the `next` command.

Example

```
% simv -ucli  
ucli% run 1000 ns  
  
ucli% next  
i8253.s, 120 : #250 CLK0 = 0;
```

step

You can use the step command to advance the simulation to the next executable line.

Example

```
ucli% step  
  
i8253.s, 119 : #150 CLK0 = 1;  
ucli% step
```

Dumping a VPD File

```
dump -add list_of_paths | objects  
[-depth levels] [-aggregates]  
[-autoflush on] [-interval seconds]  
[-file filename] [-fid fileID] [-close fileID]
```

Dumps values of the specified signals/variables/nets/regs in file *filename*. If no filename is specified, it dumps in the *inter.vpd* file.

Example

The command line to dump the complete design is:

```
% simv -ucli  
ucli% dump -add .  
1  
ucli% run
```

The above command line dumps the complete design into the default file "inter.vpd". Also, it assumes the default value for -depth as "0". However, you can specify a required level number, and VCS dumps only the specified number of levels to the VPD file. For example:

```
% simv -ucli  
ucli% dump -add . -depth 2  
1
```

The above command line dumps only the first two levels from the top into the VPD file.

In all the above example command lines, the default VPD file created is the `inter.vpd` file. To name the VPD file, use the command:

```
ucli% dump -file file_name
```

The above command line creates a dump file of type VPD with the specified filename. This command returns `fid_name`. Now, to dump the signals/scopes to this VPD file, use the returned `fid_name` with `dump -add` command as shown in the example below:

```
% simv -ucli  
ucli% dump -file alu.vpd  
VPD0  
ucli% dump -add . -fid VPD0  
1
```

The above command line dumps the entire design into `alu.vpd`.

Setting Breakpoints

```
stop [-line line_no [-file file_name]  
[-instance path] [-thread tid | allthreads] ] |  
[-absolute | -relative time] |  
[-posedge | -negedge | -change path_name] |  
[-in task/function_name | -thread tid] |  
[-show stop_id] |  
[-delete stop_id] |  
[-enable|-disable stop_id]
```

Using the `stop` command, you can perform the following:

- You can set line, time, task/function, and event breakpoints
- List all set breakpoints
- Enable, disable or delete a breakpoint

Note:

To set breakpoints, you should compile the design using the `-debug_all` option.

The following example demonstrates the above mentioned tasks.

Example1: To set line, time and event breakpoints

```
ucli% stop -file time_block.vhd -line 50
1
ucli% stop -absolute 20 ns
2
ucli% stop -change
tb_top_mix_vlog.UUT.T_BLOCK.TSM.TIME_BUTTON
3
ucli% stop
1: -line 50 -file time_block.vhd
2: -absolute 20000
3: -change tb_top_mix_vlog.UUT.T_BLOCK.TSM.TIME_BUTTON
```

This example demonstrates, setting line, time and event breakpoints. It also shows the use of `stop` command without any options to show all set breakpoints.

When you set a breakpoint, the `stop` command returns the `stop_id` as shown below:

```
ucli% stop -file time_block.vhd -line 50
1
```

In the above example, `stop_id` for the line breakpoint is 1, for the time breakpoint is 2, and for the event breakpoint is 3.

This `stop_id` can be used to disable or enable the breakpoint as shown in below:

```
ucli% stop -disable 1
1
```

Forcing and Releasing Signals

UCLI allows you to force or release a signal, wire, or a reg. The force and release commands are:

- `force`
- `release`

Force

The `force` command allows you to force a value onto a signal, wire or a reg. Activity in the tool does not override this value unless `-deposit` is specified.

The syntax is as follows:

```
force path value
      [-deposit]
      [time [ , value time] * [ -repeat delay ]]
      [-cancel time ]
```

Here:

`-deposit`

Sets the variable or signal value in an optional specified time window until a subsequent event occurs or until a `release` or another `force` is issued.

*time [, value time]**

Specifies the relative time and the time unit, at which the specified value is forced on a signal/variable/reg/wire. Use @ sign to specify the time as absolute time.

You can also specify multiple values at different times, and the specified values will be forced on the specified signal at the specified time.

-repeat delay

Forces the specified signal with the same specified value every relative time interval.

-cancel delay

Cancel the applied force on the specified signal at the specified time.

path

Specify the path to a signal/variable/reg/wire.

value

Specify the value to be forced.

Only literal values of the appropriate type can be specified for a given HDL object.

Examples

```
ucli% force top.i1.w 1'b0
```

This command immediately forces the HDL object `top.i1.w` to hold the value `1'b0`.

```
ucli% force top.i1.w 1'b0 @ 10ns
```

This command forces the value of `top.i1.w` to `1'b0` at time `10ns` of the simulation time. The "`@`" sign is optional and indicates that the following time expression is relative to the beginning of the simulation. If it is omitted, the time expression is considered as relative to the current simulation time.

```
ucli% force -deposit top.i1.w 1'b0 10ns
```

In this command, the use of `-deposit` will retain the `1'b0` on `top.i1.w` until a subsequent event occurs or a `release` or another `force` is issued.

```
ucli% force top.clk 1 10, 0 20 -repeat 30
```

Assuming the current simulation time is `0`, `top.clk` will be forced to hold the value `1` at time `10` and will hold until time `20` when it is forced to hold the value `0`. The sequence repeats every `10` time units infinitely until the effect of the `force` command is cancelled.

Release

The `release` command releases a signal/variable/net/reg from the value assigned previously using the `force` command.

The syntax is as follows:

```
release path
```

Here:

`path`

`path` specifies the path to a signal/variable/reg/wire whose force is to be released.

Example

```
ucli% force top.f1.b 15
ucli%
ucli%
... later in session
ucli%
ucli% release top.f1.b
```

Calling System Task or Function

You can call any system task or a function using the `call` command. The syntax is as follows:

```
call {tool_routine}
```

tool_routine can be a system task or a function.

Example

```
ucli% call {$vcpluson($top.dut, 0)}
```

Macro Control Routine

Using the `do` command you can execute any Tcl script at the UCLI prompt. The syntax is as follows:

```
do macro_script
```

Example

```
ucli% do my.tcl
Start Simulation

Writing Mode #0 -- LSB Load, Mode 0, Binary

End Simulation
      V C S   S i m u l a t i o n   R e p o r t
Time: 100 ns
CPU Time:    0.000 seconds;      Data structure size:  0.9Mb
Wed Sep 19 10:49:28 2007
```

The contents of my.tcl is:

```
% cat my.tcl
puts "Start Simulation"
run 100
puts "End Simulation"
quit
```

Navigation Command

scope

The `scope` command shows the current scope you are in. You can also use the `scope` command to move to a different scope. The syntax is as follows:

```
scope path_name [-active]
```

Here:

`path_name`

Changes the current scope to the specified relative or the absolute path.

-active

Change the current scope to the active scope.

Example:

```
% simv -ucli  
  
ucli> scope  
/TOP/U_DESIGN/VLOG_DUT
```

Design Query Commands

```
show [-instances] | [-value] | [-type] |  
[-radix [binary | decimal | octal | hexadecimal]]  
[-mailbox] | [-semaphore] [path_name]
```

You can use the `show` command with the following options:

- `-instances` to get the list of instances in the specified instance.

Example:

```
ucli% show -instances UUT  
RSM  
CONVERT  
COMPARE
```

- `-value` to get the current value of the specified signal/wire/reg.

Example:

```
ucli% show -value CLK  
CLK 0
```

- `-type` to get the type of the specified object.

Example:

```
ucli% show -type CLK
CLK {BASE {} reg}
```

- `-radix [binary | decimal | octal | hexadecimal]` to show the value of the specified signal in the specified radix. The `-radix` option should be used along with the `-value` option.

Example:

```
ucli% show -value CLK -radix hexadecimal
CLK 'h0
```

- `-mailbox` to show the specified mailbox or all mailboxes and shows the data/blocked threads.

Example:

```
ucli% show -mailbox
mailbox 1: data(2): -->5 -->15
mailbox 2: blocked threads: 3, 4
```

- `-semaphore` to show the specified semaphore or all semaphores and the #keys and/or blocked threads.

Example:

```
ucli% show -semaphore
semaphore 1: keys(5) blocked threads: 5, 6
```

drivers *path_name* [-verbose|-local]

drivers displays the drivers of the specified object *path_name*.

Example:

```
ucli% driver CLK -verbose
0 - reg tb.CLK
  0 tb.CLK /u/vlog_top/2layer/tb.v 58 :      CLK = 1'b0;
  0 tb.CLK /u/vlog_top/2layer/tb.v 75 :      CLK = 1'b1;
```


10

Performance Tuning

VCS delivers the best performance during both compile-time and runtime by reducing the size of the simulation executable, and the amount of memory consumed for compilation and simulation. By default, it is optimized for the following types of designs:

- Designs with many layers of hierarchy
- Gate-level designs
- Structural RTL-level designs - Using libraries where the cells are RTL-level code
- Designs with extensive use of timing such as delays, timing checks, and SDF back annotation, particularly to INTERCONNECT delays

However, depending on the phase of your design cycle, you can fine-tune VCS for a better compile-time and runtime performance.

This chapter describes the following sections:

- Compile-time Performance

Compile-time performance plays a very important role when you are in the initial phase of your design development cycle. In this phase, you may want to modify and recompile the design to observe the behavior. Since, this phase involves lot many recompiling cycles, achieving a faster compilation is important. For additional information, see the section entitled, “[Compile-time Performance](#)”.

- Runtime Performance

Runtime performance is important in regression phase or in the final phase of the design development cycle. For additional information, see the section entitled, “[Runtime Performance](#)”.

Compile-time Performance

You can improve compile-time performance in the following ways:

- “[Incremental Compilation](#)”
- “[Compile Once and Run Many Times](#)”
- “[Parallel Compilation](#)”

Incremental Compilation

During compilation, VCS builds the design hierarchy. By default, when you recompile the design, VCS compiles only those design units that have changed since the last compilation. This is called incremental compilation.

The incremental compilation feature is the default in VCS. It triggers recompilation of design units under the following conditions:

- Changes in the command-line options.
- Change in the target of a hierarchical reference.
- Change in the ports of a design unit.
- Change in the functional behavior of the design.
- Change in a compile-time constant such as a parameter.

The following conditions do not cause VCS to recompile a module:

- Change of time stamp of any source file.
- Change in file name or grouping of modules in any source file.
- Unrelated change in the same source file.
- Nonfunctional changes such as comments or white space.

Compile Once and Run Many Times

The VCS usage model is devised in such a way that you can create a single binary executable and execute it many times avoiding the elaboration step for all but the first run. For information on the VCS usage model, see “[Using VCS](#)” on page 1-8.

For example, you can use this feature in the following scenarios:

- Use VCS runtime features, like passing values at runtime, to modify the design, and simulate it without re-compiling. For information on runtime features, see [Chapter 5, "Simulating the Design"](#).
- Run the same test with different seeds.
- Create a softlink of the executable and the .daidir or .db.dir directory in a different directory, to run multiple simulations in parallel.

Parallel Compilation

You can improve the compile-time performance by specifying the number of parallel processes VCS can launch for the native code generation phase of the compilation. You should specify this using the compile-time option `-j [no_of_processes]`, as shown below:

```
% vcs -j [no_of_processes] [options] top_entity/module/  
config
```

For example, the following command line will fork off two parallel processes to generate a binary executable:

```
% vcs -j2 top
```

Runtime Performance

VCS runtime performance is based on the following:

- Coding Style (see Chapter 3, "Modeling Your Design").
- Access to the internals of your design at runtime, using PLIs, UCLI, debugging using GUI, dumping waveforms etc.

This section describes the following to improve the runtime performance:

- ["Using R radiant Technology"](#)
- ["Improving Performance When Using PLIs"](#)

Using R radiant Technology

VCS's R radiant Technology applies performance optimizations to the Verilog portion of your design while VCS compiles your Verilog source code. These R radiant optimizations improve the simulation performance of all types of designs from behavioral, RTL to gate-level designs. R radiant Technology particularly improves the performance of functional simulations where there are no timing specifications or when delays are distributed to gates and assignment statements.

Compiling With R radiant Technology

R radiant Technology optimizations are not enabled by default. You enable them using the compile-time options:

`+rad`

Specifies using Radiant Technology

`+optconfigfile`

Optional. Specifies applying Radiant Technology optimizations to part of the design using a configuration file as described below:

Applying Radiant Technology to Parts of the Design

The configuration file enables you to apply Radiant optimizations selectively to different parts of your design. You can enable or disable Radiant optimizations for all instances of a module, specific instances of a module, or specific signals.

You specify the configuration file with the `+optconfigfile` compile-time option. For example:

`+optconfigfile+file_name`

Note:

The configuration file is a general purpose file that has other purposes, such as specifying ACC write capabilities. Therefore, to enable Radiant Technology optimizations with a configuration file, you must also include the `+rad` compile-time option.

The Configuration File Syntax

The configuration file contains one or more statements that set Radiant optimization attributes, such as enabling or disabling optimization on a type of design object, such as a module definition, a module instance, or a signal.

The syntax of each type of statement is as follows:

module {*list_of_module_identifiers*} {*list_of_attributes*} ;

or

instance

{*list_of_module_identifiers_and_hierarchical_names*}
{*list_of_attributes*} ;

or

tree [(*depth*)] {*list_of_module_identifiers*}
{*list_of_attributes*} ;

Usage:

module

Keyword that specifies that the attributes in this statement apply to all instances of the modules in the list, specified by module identifier.

list_of_module_identifiers

A comma separated list of module identifiers enclosed in curly braces: { }

list_of_attributes

A comma separated list of Radiant optimization attributes enclosed in curly braces: { }

instance

Keyword that specifies that the attributes in this statement apply to:

- All instances of the modules in the list specified by module identifier.
- All module instances in the list specified by their hierarchical names and all the other instances as well. VCS determines the module definition for each module instance specified and applies the attributes to all instances of the module not just the specified module instance.
- The individual signals in the list specified by their hierarchical names.

`list_of_module_identifiers_and_hierarchical_names`

A comma separated list of module identifiers and hierarchical names of module instances and signals enclosed in curly braces: { }

Note:

Follow the Verilog syntax for signal names and hierarchical names of module instances.

`tree`

Keyword that specifies that the attributes in this statement apply to all instances of the modules in the list, specified by module identifier, and also apply to all module instances hierarchically under these module instances.

depth

An integer that specifies how far down the module hierarchy, from the specified modules, you want to apply Radiant optimization attributes. You can specify a negative value. A negative value specifies descending to the leaf level and counting up levels of the hierarchy to apply these attributes. This specification is optional. Enclose this specification in parentheses: ()

The valid Radiant optimization attributes are as follows:

noOpt

Disables Radiant optimizations on the module instance or signal.

noPortOpt

Prevents port optimizations such as optimizing away unused ports on a module instance.

Opt

Enables all possible Radiant optimizations on the module instance or signal.

PortOpt

Enables port optimizations such as optimizing away unused ports on a module instance.

Statements can use more than one line and must end with a semicolon.

Verilog style comments characters /* *comment* */ and // *comment* can be used in the configuration file.

Configuration File Statement Examples

The following are examples of statements in a configuration file.

module statement example

```
module {mod1, mod2, mod3} {noOpt, PortOpt};
```

This module statement example disables Radiant optimizations for all instances of modules mod1, mod2, and mod3, with the exception of port optimizations.

multiple module statement example

```
module {mod1, mod2} {noOpt};  
module {mod1} {Opt};
```

In this example, the first module statement disables radiant optimizations for all instances of modules mod1 and mod2 and then the second module statement enables Radiant optimizations for all instances of module mod1. VCS processes statements in the order in which they appear in the configuration file so the enabling of optimizations for instances of module mod1 in the second statement overrides the first statement.

instance statement example

```
instance {mod1} {noOpt};
```

In this example, mod1 is a module identifier so the statement disables Radiant optimizations for all instances of mod1. This statement is the equivalent of:

```
module {mod1} {noOpt};
```

module and instance statement example

```
module {mod1} {noOpt};  
instance {mod1.mod2_inst1.mod3_inst1,
```

```
mod1.mod2_inst1.reg_a} {noOpt};
```

In this example, the module statement disables Radiant optimizations for all instances of module mod1.

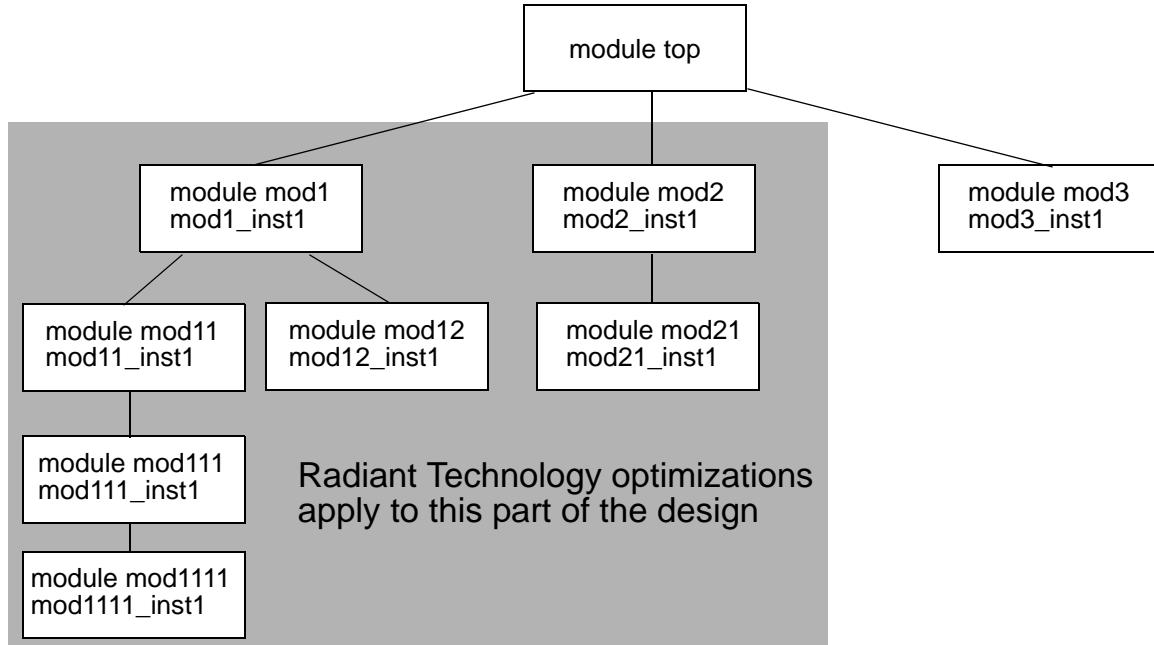
The instance statement disables Radiant optimizations for the following:

- Module mod1 (already disabled by the module statement)
- The module instance with the instance identifier mod2_inst1 in mod1
- The module instance with the instance identifier mod3_inst1 under module instance mod2_inst1
- Signal reg_a in module instance mod2_inst1

first tree statement example

```
tree {mod1,mod2} {Opt};
```

This example is for a design with the following module hierarchy:

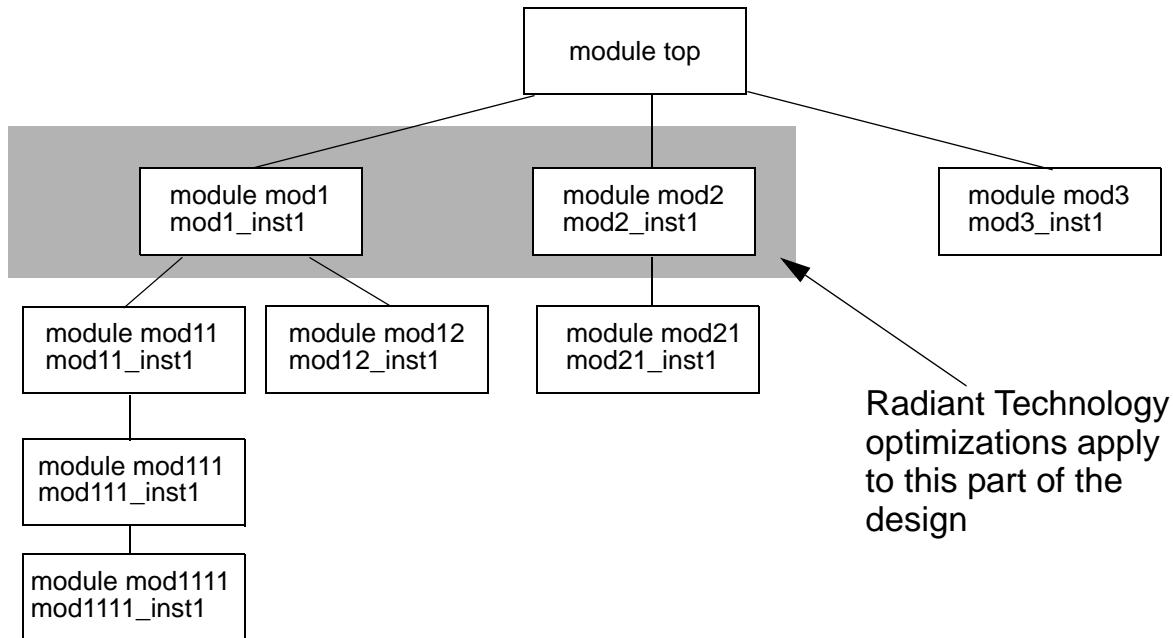


The statement enables Radiant Technology optimizations for the instances of modules `mod1` and `mod2` and for all the module instances hierarchically under these instances.

second tree statement example

```
tree (0) {mod1,mod2} {Opt};
```

This modification of the previous tree statement includes a depth specification. A depth of 0 means that the attributes apply no further down the hierarchy than the instances of the specified modules, mod1 and mod2.



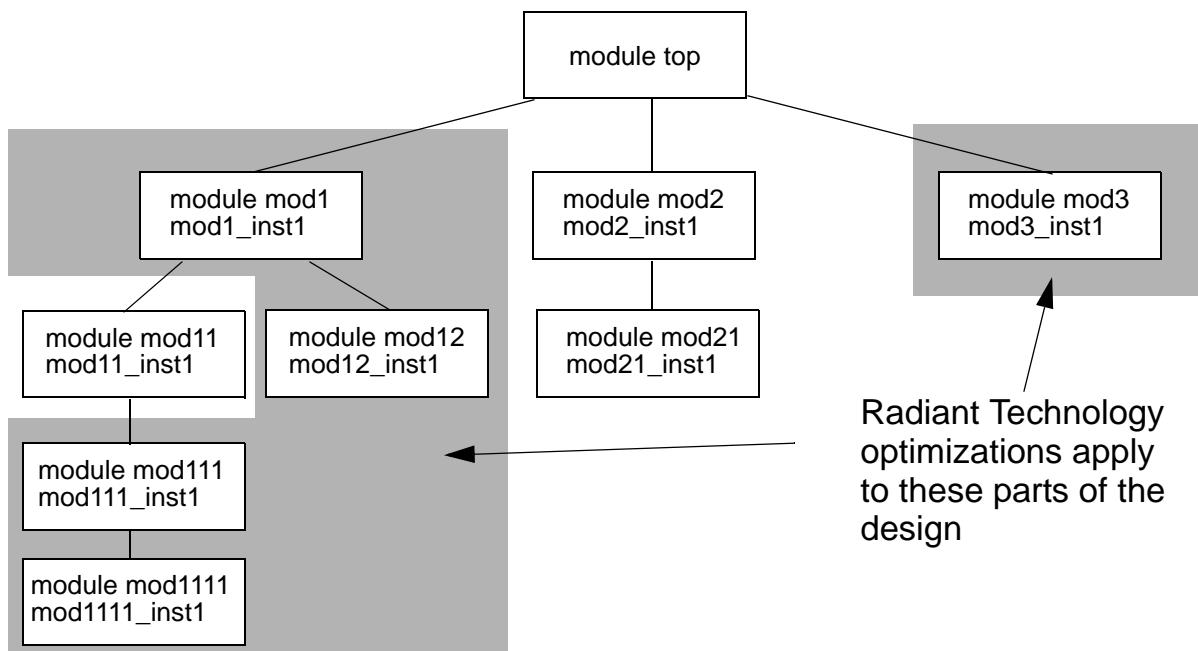
A tree statement with a depth of 0 is the equivalent of a module statement.

third tree statement example

You can specify a negative value for the depth value. If you do this, specify ascending the hierarchy from the leaf level. For example:

```
tree (-2) {mod1, mod3} {Opt};
```

This statement specifies looking down the module hierarchy under the instances of modules `mod1` and `mod3` to the leaf level and counting up from there. (Leaf level module instances contain no module instantiation statements.)



In this example, the instances of `mod1111`, `mod12`, and `mod3` are at a depth of -1 and the instances of `mod111` and `mod1` are at a depth of -2. The attributes do not apply to the instance of `mod11` because it is at a depth of -3.

fourth tree statement example

You can disable Radiant optimizations at the leaf level under specified modules. For example:

```
tree(-1) {mod1, mod2} {noOpt};
```

This example disables optimizations at the leaf level, the instances of modules mod1111, mod12, and mod21, under the instances of modules mod1 and mod2.

Known Limitations

Radiant Technology is not applicable to all simulation situations. Some features of VCS are not available when you use Radiant Technology.

These limitations are:

- Back-annotating SDF Files

You cannot use Radiant Technology if your design back-annotates delay values from either a compiled or an ASCII SDF file at runtime.

- SystemVerilog

Radiant Technology does not work with SystemVerilog design construct code. For example, structures and unions, new types of always blocks, interfaces, or things defined in \$root.

The only SystemVerilog constructs that work with Radiant Technology are SystemVerilog assertions that refer to signals with Verilog-2001 data types, not the new data types in SystemVerilog.

Potential Differences in Coverage Metrics

VCS supports coverage metrics with Radiant Technology and you can enter both the `+rad` and `-cm` compile-time options. However, Synopsys does not recommend comparing coverage between two simulation runs when only one simulation was compiled for Radiant Technology.

The Radiant Technology optimizations, though not changing the simulation results, can change the coverage results.

Compilation Performance With Radiant Technology

Using Radiant Technology incurs longer incremental compile times because the analysis performed by Radiant Technology occurs every time you recompile the design even when only a few modules have changed. However, VCS only performs the code generation phase on the parts of the design that have actually changed. Therefore, the incremental compile times are longer when you use Radiant Technology but shorter than a full recompilation of the design.

Improving Performance When Using PLIs

As mentioned earlier, the runtime performance is reduced when you have PLIs accessing the design. In some cases, you may have ACC capabilities enabled on all the modules in the design, including those which actually do not require them. These scenarios will unnecessarily reduce the runtime performance. Ideally the performance can be improved if you are able to control the access rights of the PLIs. However, this may not be possible in many situations. In this situation, you can use the `+vcs+learn+pli` runtime option.

`+vcs+learn+pli` tells VCS to write a new tab file with the ACC capabilities enabled on the modules/scopes which actually need them during runtime. Now, during recompile, along with your original tab file, you can pass the new tab file using the compile-time option, `+applylearn+ [tabfile]`, so that the next simulation will have a better runtime. Therefore, this is a two-step process:

- Using the runtime option `+vcs+learn+pli`
- Using the compilation option `+applylearn+ [tabfile]` during recompile. You do not have to reanalyze the files in this step.

The usage model and an example is shown below:

Usage Model

Step1: Using the runtime option `+vcs+learn+pli`.

Compilation

```
% vcs [vcs_options] Verilog_files
```

Simulation

```
% simv [sim_options] +vcs+learn+pli
```

Step2: Using the compilation option +applylearn+ [*tabfile*].

Compilation

```
% vcs [vcs_options] +applylearn+ [tabfile] Verilog_files
```

Simulation

```
% simv [sim_options]
```

Impact on Performance

Options like -debug_pp, -debug, and -debug_all disable VCS optimizations and also impact the performance. The -debug_pp option has less performance impact than the -debug or -debug_all options. The following table describes these options and their performance impact:

Table 10-1 Performance Impact of -debug_pp, -debug, and -debug_all

Options	Description
-debug_pp	Use this option to generate a dump file. You can also use this option to invoke UCLI and DVE with some limitations. This has less performance impact when compared to -debug or -debug_all
-debug	Use this option if you want to use the force command at the UCLI prompt, and for more debug capabilities.
-debug_all	This option enables all debug capabilities, and therefore will have a huge performance impact.

See the section “[Compiling the Design in Debug Mode](#)” on page 4-1 for more information.

Note that using extensive user interface commands, like force or release at runtime, will have an enormous impact on the performance. To improve the performance, Synopsys recommends you convert these user interface commands to HDL files and to compile and simulate them along with the design.

Contact Synopsys Support Center (supt@synopsys.com) or your Synopsys Application Consultant for further assistance.

Performance Tuning

10-20

11

Gate-level Simulation

This chapter describes the following sections:

- “SDF Annotation”
- “Delays and Timing”
- “Using the Configuration File to Disable Timing”
- “Using the timopt Timing Optimizer”
- “Negative Timing Checks”

SDF Annotation

The OVI Standard Delay File (SDF) specification provides a standard ASCII file format for representing and applying delay information. VCS supports the OVI versions 1.0, 1.1, 2.0, 2.1, and 3.0 of this specification.

In the SDF format a tool can specify intrinsic delays, interconnect delays, port delays, timing checks, timing constraints, and pulse control (PATHPULSE).

When VCS reads an SDF file it “back-annotates” delay values to the design, that is, it adds delay values or changes the delay values specified in the source files.

Using \$sdf_annotation System Task

You can use the \$sdf_annotation system task to back-annotate delay values from an SDF file to your Verilog design.

The syntax for the \$sdf_annotation system task is as follows:

```
$sdf_annotation ("sdf_file" [, module_instance]
 [, "sdf_configfile"] [, "sdf_logfile"] [, "mtm_spec"]
 [, "scale_factors"] [, "scale_type"]);
```

Where:

"sdf_file"

Specifies the path to the SDF file.

`module_instance`

Specifies the scope where back-annotation starts. The default is the scope of the module instance that calls `$sdf_annotation`.

`"sdf_configfile"`

Specifies the SDF configuration file.

`"sdf_logfile"`

Specifies the SDF log file to which VCS sends error messages and warnings. By default, VCS displays no more than ten warning and ten error messages about back-annotation and writes no more than that in the log file you specify with the `-l` option. However, if you specify an SDF log file with this argument, the SDF log file receives all messages about back-annotation. You can also use the `+sdfverbose` runtime option to enable the display of all back-annotation messages.

`"mtm_spec"`

Specifies which delay values of `min:typ:max` triplets VCS back-annotes. Specify MINIMUM, TYPICAL, MAXIMUM or TOOL_CONTROL (default).

`"scale_factors"`

Specifies the multiplier for the minimum, typical and maximum components of delay triplets. It is a colon separated string of three positive, real numbers "1.0:1.0:1.0" by default.

`"scale_type"`

Specifies the delay value from each triplet in the SDF file for use before scaling. Possible values: "FROM_TYPICAL", "FROM_MIMINUM", "FROM_MAXIMUM", "FROM_MTM" (default).

The usage model to simulate a design using `$sdf_annotate` is the same as the basic usage model as shown below:

Compilation

```
% vcs [elab_options] top_cfg/entity/module
```

Simulation

```
% simv [run_options]
```

See “[Options for Specifying Delays and SDF File](#)” on page B-29.

Delays and Timing

This section describes the following topics:

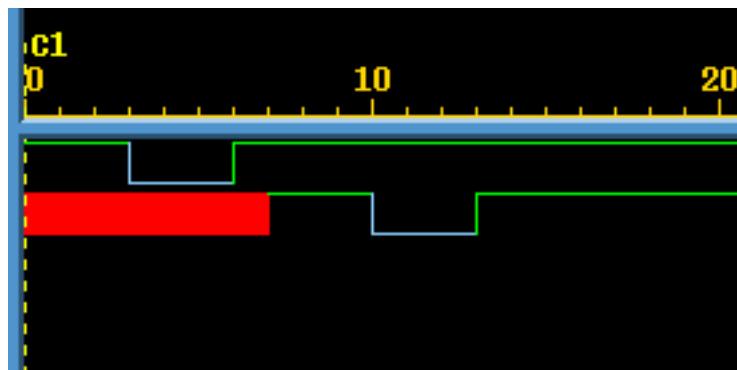
- [“Transport and Inertial Delays”](#)
- [“Pulse Control”](#)
- [“Specifying the Delay Mode”](#)

Transport and Inertial Delays

Delays can be categorized into transport and inertial delays.

Transport delays allow all pulses that are narrower than the delay to propagate through. For example, [Figure 11-1](#) shows the waveforms for an input and output port of a module that models a buffer with a module path delay of seven time units between these ports. The waveform on top is that of the input port and the waveform underneath is that of the output port. In this example, you have enabled transport delays for module path delays and specified that a pulse three time units wide can propagate through. For an explanation on how this is done, see “[Enabling Transport Delays](#)” on [page 11-9](#) and “[Pulse Control](#)” on [page 11-10](#).

Figure 11-1 Transport Delay Waveforms

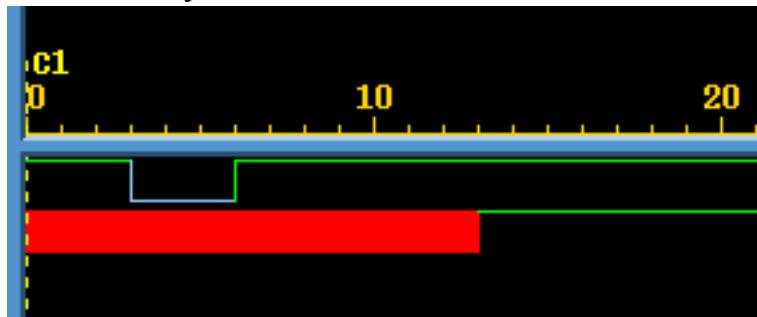


At time 0, a pulse three time units wide begins on the input port. This pulse is narrower than the module path delay of seven time units, but this pulse propagates through the module and appears on the output port after seven time units. Similarly, another narrow pulse begins on the input port at time 3 and it also appears on the output port seven time units later.

You can apply transport delays on all module path delays and all SDF INTERCONNECT delays back-annotated to a net from an SDF file. For more information on SDF back-annotation, see [Chapter 20, "Using SystemVerilog Assertions"](#).

Inertial delays, in contrast, filter out all pulses that are narrower than the delay. [Figure 11-2](#) shows the waveforms for the same input and output ports when you have not enabled transport delays for module path delays.

Figure 11-2 Inertial Delay Waveforms



The pulse that begins at time 0 that is three time units wide does not propagate to the output port because it is narrower than the seven time unit module path delay. Neither does the narrow pulse that begins at time 3. Note that the wide pulse that begins at time 6 does propagate to the output port.

Gates, switches, MIPDs, and continuous assignments only have inertial delays, which are the default type of delay for module path delays and INTERCONNECT delays back-annotated from an SDF file to a net.

Different Inertial Delay Implementations

For compatibility with the earlier generation of Verilog simulators, inertial delays have two different implementations, one for primitives (gates, switches and UDPs), continuous assignments, and MIPDs

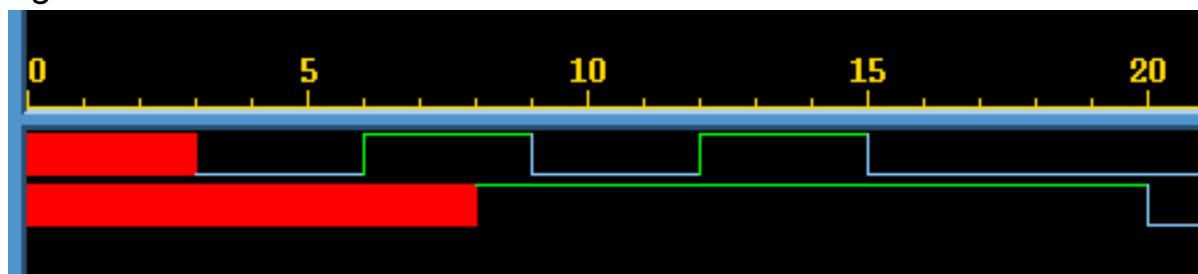
(Module Input Port Delays) and the other for module path delays and INTERCONNECT delays back-annotated from an SDF file to a net. For more details on SDF back-annotation, see [Chapter 20, "Using SystemVerilog Assertions"](#). There is also a third implementation that is for module path and INTERCONNECT delays and pulse control, see [“Pulse Control” on page 11-10](#).

Inertial Delays for Primitives, Continuous Assignments, and MIPDs

Both implementations were devised to filter out narrow pulses but the one for primitives, continuous assignments, and MIPDs can produce unexpected results. For example, [Figure 11-3](#) shows the waveforms for nets connected to the input and output terminals of a buf gate with a delay of five time units.

In this implementation there can never be more than one scheduled event on an output terminal. To filter out narrow pulses, the trailing edge of a pulse can alter the value change but not the transition time of the event scheduled by the leading edge of the pulse if the event has not yet occurred.

Figure 11-3 Gate Terminal Waveforms



In the example illustrated in [Figure 11-3](#), the following occurs:

1. At time 3 the input terminal changes to 0. This is the leading edge of a three time unit wide pulse. This event schedules a value change to 0 on the output terminal at time 8 because there is a #5 delay specification for the gate.
2. At time 6 the input terminal toggles to 1. This implementation keeps the scheduled transition on the output terminal at time 8 but alters the value change to a value of 1.
3. At time 8 the output terminal transitions to 1. This transition might be unexpected because all pulses on the input have been narrower than the delay, but this is how this implementation works. There is now no event scheduled on the output and a new event can now be scheduled.
4. At time 9 the input terminal toggles to 0 and the implementation schedules a transition of the output to 0 at time 14.
5. At time 12 the input terminal toggles to 1 and the value change scheduled on the output at time 14 changes to a 1.
6. At time 14 the output is already 1 so there is no value change. The narrow pulse on the input between time 9 and 12 is filtered out. This implementation was devised for these narrow pulses. There is now no event scheduled for the output.
7. At time 15 the input toggles to 0 and this schedules the output to toggle to 0 at time 20.

Inertial Delays for Module Path Delays and INTERCONNECT Delays

The implementation of inertial delays for module path delays and SDF INTERCONNECT delays is as follows: if the event scheduled by the leading edge of a pulse is scheduled for a later simulation time, or in other words, has not yet occurred, then the event

scheduled by the trailing edge at the end of the specified delay and at a new simulation time, replaces the event scheduled by the leading edge. All narrow pulses are filtered out.

Note:

- SDF INTERCONNECT delays follow this implementation if you include the `+multisource_int_delays` compile-time option. If you do not include this option, VCS uses an MIPD to model the SDF INTERCONNECT delay and the delay uses the inertial delay implementation for MIPDs.
- VCS enables more complex and flexible pulse control processing when you include the `+pulse_e/number` and `+pulse_r/number` options. See “[Pulse Control](#)” on page 11-10.

Enabling Transport Delays

Transport delays are never the default delay.

You can specify transport delays on module path delays with the `+transport_path_delays` compile-time option. For this option to work, you must also include the `+pulse_e/number` and `+pulse_r/number` compile-time options. See “[Pulse Control](#)” on page 11-10.

You can specify transport delays on a net to which you back-annotate SDF INTERCONNECT delays with the `+transport_int_delays` compile-time option. For this option to work, you must also include the `+pulse_int_e/number` and `+pulse_int_r/number` compile-time options. See “[Pulse Control](#)” on page 11-10.

The `+pulse_e/number`, `+pulse_r/number`, `+pulse_int_e/number`, and `+pulse_int_r/number` options define specific thresholds for pulse width, which allow you to tell VCS to filter out only some of the pulses and let the other pulses through. See “[Pulse Control](#)” on page 11-10.

Pulse Control

So far we've seen that with pulses narrower than a module path or INTERCONNECT delay, you have the option of filtering all of them out by using the default inertial delay or allowing all of them to propagate through, by specifying transport delays. VCS also provides a third option - pulse control. With pulse control you can:

- Allow pulses that are slightly narrower than the delay to propagate through.
- Have VCS replace even narrower pulses with an x value pulse on the output and display a warning message.
- Have VCS then filter out and ignore pulses that are even narrower than the ones for which it propagates an x value pulse and displays an error message.

You specify pulse control with the `+pulse_e/number` and `+pulse_r/number` compile-time options for module path delays and the `+pulse_int_e/number` and `+pulse_int_r/number` compile-time options for INTERCONNECT delays.

The `+pulse_e/number` option's *number* argument specifies a percentage of the module path delay. VCS replaces pulses whose widths that are narrower than the specified percentage of the delay with an x value pulse on the output or inout port and displays a warning message.

Similarly, the `+pulse_int_e/number` option's *number* argument specifies a percentage of the INTERCONNECT delay. VCS replaces pulses whose widths are narrower than the specified percentage of the delay with an `x` value pulse on the inout or output port instance that is the load of the net to which you back-annotated the INTERCONNECT delay. It also displays a warning message.

The `+pulse_r/number` option's *number* argument also specifies a percentage of the module path delay. VCS filters out the pulses whose widths are narrower than the specified percentage of the delay. With these pulses there is no warning message; VCS simply ignores these pulses.

Similarly, the `+pulse_int_r/number` option's *number* argument specifies a percentage of the INTERCONNECT delay. VCS filters out pulses whose widths are narrower than the specified percentage of the delay. There is no warning message with these pulses.

You can use pulse control with transport delays (see “[Pulse Control with Transport Delays](#)” on page 11-12) or inertial delays (see “[Pulse Control with Inertial Delays](#)” on page 11-14).

When a pulse is narrow enough for VCS to display a warning message and propagate an `x` value pulse, you can set VCS to do one of the following:

- Place the starting edge of the `x` value pulse on the output, as soon as it detects that the pulse is sufficiently narrow, by including the `+pulse_on_detect` compile-time option.
- Place the starting edge on the output at the time when the rising or falling edge of the narrow pulse would have propagated to the output. This is the default behavior.

See “[Specifying Pulse on Event or Detect Behavior](#)” on page 11-18.

Also when a pulse is sufficiently narrow to display a warning message and propagate an x value pulse, you can have VCS propagate the x value pulse but disable the display of the warning message with the `+no_pulse_msg` runtime option.

Pulse Control with Transport Delays

You specify transport delays for module path delays with the `+transport_path_delays`, `+pulse_e/number`, and `+pulse_r/number` options. You must include all three of these options.

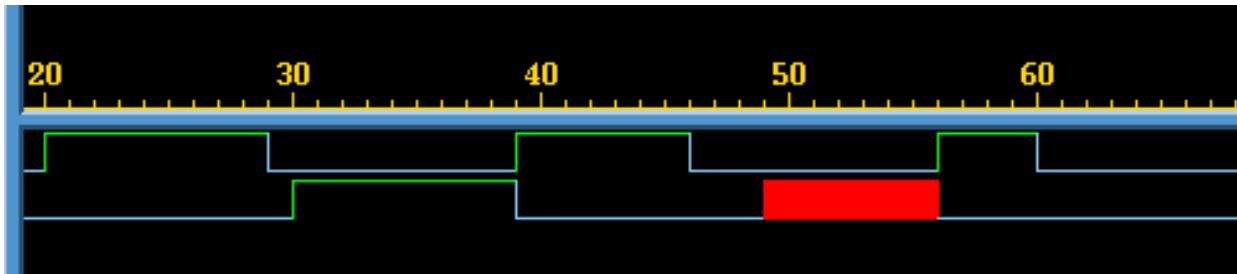
You specify transport delays for INTERCONNECT delays on nets with the `+transport_int_delays`, `+pulse_int_e/number`, and `+pulse_int_r/number` options. You must include all three of these options.

If you want VCS to propagate all pulses, no matter how narrow, specify a 0 percentage. For example, if you want VCS to replace pulses that are narrower than 80% of the delay with an x value pulse (and display a warning message) and filter out pulses that are narrower than 50% of the delay, enter the `+pulse_e/80` and `+pulse_r/50` or `+pulse_int_e/80` and `+pulse_int_r/50` compile-time options.

[Figure 11-4](#) shows the waveforms for the input and output ports for an instance of a module that models a buffer with a ten time unit module path delay. The `vcs` command line contains the following compile-time options:

```
+transport_path_delays +pulse_e/80 +pulse_r/50
```

Figure 11-4 Pulse Control with Transport Delays



In the example illustrated in Figure 11-4 the following occurs:

1. At time 20, the input port toggles to 1.
2. At time 29, the input port toggles to 0 ending a nine time unit wide value 1 pulse on the input port.
3. At time 30, the output port toggles to 1. The nine time unit wide value 1 pulse that began at time 20 on the input port is propagating to the output port because we have enabled transport delays and nine time units is more than 80% of the ten time unit module path delay.
4. At time 39, the input port toggles to 1 ending a ten time unit wide value 0 pulse. Also, at time 39 the output port toggles to 0. The ten time unit wide value 0 pulse that began at time 29 on the input port is propagating to the output port.
5. At time 46, the input port toggles to 0 ending a seven time unit wide value 1 pulse.
6. At time 49, the output port transitions to x. The seven time unit wide value 1 pulse that began at time 39 on the input port has propagated to the output port, but VCS has replaced it with an x value pulse because seven time units is less than 80% of the module path delay. VCS issues a warning message in this case.

7. At time 56, the input port toggles to 1 ending a ten time unit wide value 0 pulse. Also, at time 56, the output port toggles to 0. The ten time unit wide value 0 pulse that began at time 46 on the input port is propagating to the output port.
8. At time 60, the input port toggles to 0 ending a four time unit wide value 1 pulse. Four time units is less than 50% of the module path delay, therefore, VCS filters out this pulse and no indication of it appears on the output port.

Pulse Control with Inertial Delays

You can enter the `+pulse_e/number` and `+pulse_r/number` or `+pulse_int_e/number` and `+pulse_int_r/number` options without the `+transport_path_delays` or `+transport_int_delays` options. If you do this, you are specifying pulse control for inertial delays on module path delays and INTERCONNECT delays.

There is a special implementation of inertial delays with pulse control for module path delays and INTERCONNECT delays. In this implementation, value changes on the input can schedule two events on the output.

The first of these two scheduled events always causes a change on the output. The type of value change on the output is determined by the following:

- If the first event is scheduled by the leading edge of a pulse whose width is equal to or wider than the percentage specified by the `+pulse_e/number` option, the value change on the input propagates to the output.

- If the pulse is not wider than the percentage specified by the `+pulse_e/number` option, but is wider than the percentage specified by the `+pulse_r/number` option, the value change is replaced by an x value.
- If the pulse is not wider than the percentage specified by the `+pulse_r/number` option, the pulse is filtered out.

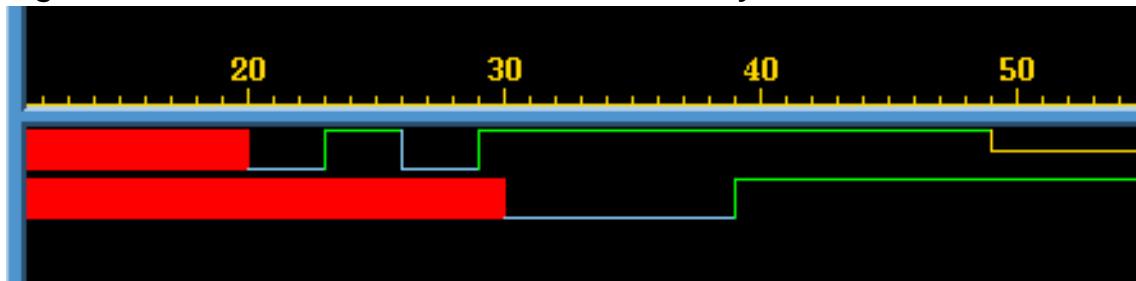
The second scheduled event is always tentative. If another event occurs on the input before the first event occurs on the output, that additional event on the input cancels the second scheduled event and schedules a new second event.

[Figure 11-5](#) shows the waveforms for the input and output ports for an instance of a module that models a buffer with a ten time unit module path delay. The `vcs` command line contains the following compile-time options:

```
+pulse_e/0 +pulse_r/0
```

In this example, specifying 0 percentages means that the trailing edge of all pulses can change the second scheduled event on the output. Specifying 0 does not mean that all pulses propagate to the output because this implementation has its own way of filtering out short pulses.

Figure 11-5 Pulse Control with Inertial Delays



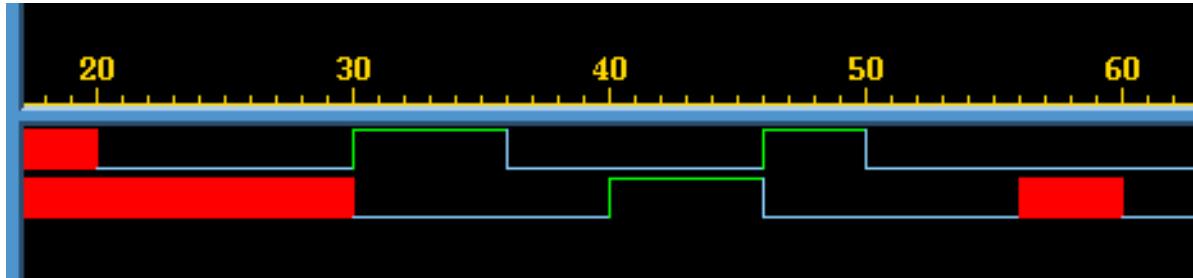
In the example illustrated in [Figure 11-5](#) the following occurs:

1. At time 20, the input port transitions to 0. This schedules a transition to 0 on the output port at time 30, ten time units later as specified by the module path delay. This is the first scheduled event on the output port. This event is not tentative, it will occur.
2. At time 23, the input port toggles to 1. This schedules a transition to 1 on the output port at time 33. This is the second scheduled event on the output port. This event is tentative.
3. At time 26, the input port toggles to 0. This cancels the current scheduled second event and replaces it by scheduling a transition to 0 at time 36. The first scheduled event is a transition to 0 at time 30 so the new second scheduled event isn't really a transition on the output port. This is how this implementation filters out narrow pulses.
4. At time 29, the input port toggles to 1. This cancels the current scheduled second event and replaces it by scheduling a transition to 1 at time 39.
5. At time 30, the output port transitions to 0. The second scheduled event on the output becomes the first scheduled event and is therefore no longer tentative.
6. At time 39, the output port toggles to 1.

Typically, however, you will want to specify that VCS replace or reject certain narrow pulses. [Figure 11-6](#) shows the waveforms for the input and output ports for an instance of the same module with a ten time unit module path delay. The `vcs` command line contains the following compile-time options:

```
+pulse_e/60 +pulse_r/40
```

Figure 11-6 Pulse Control with Inertial Delays and a Narrow Pulses



In the example illustrated in [Figure 11-6](#) the following occurs:

1. At simulation time 20, the input port transitions to 0. This schedules the first event on the output port, a transition to 0 at time 30.
2. At simulation time 30, the input port toggles to 1. This schedules the output port to toggle to 1 at time 40. Also, at simulation time 30, the output port transitions to 0. It doesn't matter which of these events happened first. At the end of this time there is only one scheduled event on the output.
3. At simulation time 36, the input port toggles to 0. This is the trailing edge of a six time unit wide value 1 pulse. The pulse is equal to the width specified with the `+pulse_e/60` option so VCS schedules a second event on the output, a value change to 0 on the output at time 46.

4. At simulation time 40, the output toggles to 1 so now there is only one event scheduled on the output, the value change to 0 at time 46.
5. At simulation time 46, the input toggles to 1 scheduling a transition to 1 at time 56 on the output. Also at time 46, the output toggles to 0. There is now only one event scheduled on the output.
6. At time 50, input port toggles to 0. This is the trailing edge of a four time unit wide value 1 pulse. The pulse is not equal to the width specified with the `+pulse_e/60` option, but is equal to the width specified with the `+pulse_r/40` option, therefore, VCS changes the first scheduled event from a change to 1 to a change to X at time 56 and schedules a second event on the output, a transition to 0 at time 60.
7. At time 56, the output transitions to X and VCS issues a warning message.
8. At time 60, the output transitions to 0.

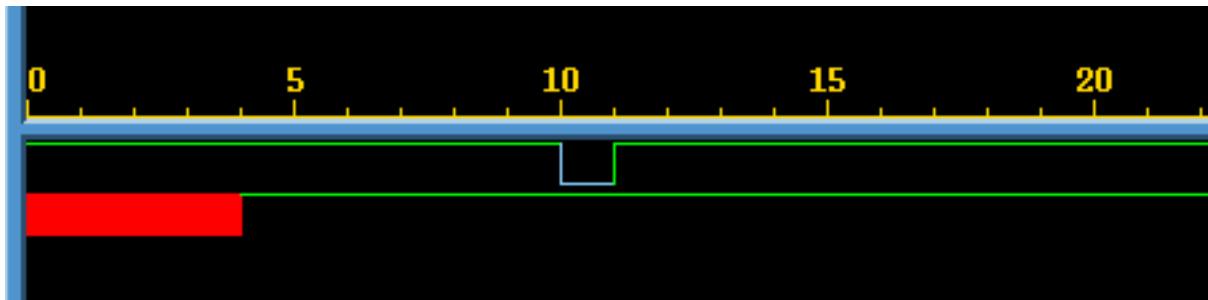
Pulse control sometimes blurs the distinction between inertial and transport delays. In this example, the results would have been the same if you also included the `+transport_path_delays` option.

Specifying Pulse on Event or Detect Behavior

Asymmetric delays, such as different rise and fall times for a module path delay, can cause schedule cancellation problems for pulses. These problems persist when you specify transport delay and can persist for a wide range of percentages that you specify for the pulse control options.

For example, for a module that models a buffer, if you specify a rise time of 4 and a fall time of 6 for a module path delay, a narrow value 0 pulse can cause scheduling problems, as illustrated in [Figure 11-7](#).

Figure 11-7 Asymmetric Delays and Scheduling Problems



In this example, you include the `+pulse_e/100` and `+pulse_r/0` options. The scheduling problem is that the leading edge of the pulse on the input, at time 10, schedules a transition to 0 on the output at time 16; but the trailing edge, at time 11, schedules a transition to 1 on the output at time 15.

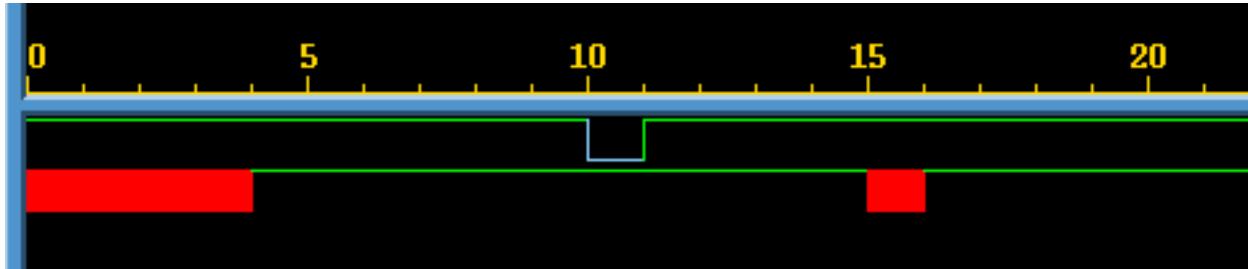
Obviously, the output has to end up with a value of 1 so VCS can't allow the events scheduled at time 15 and 16 to occur in sequence; if it did, the output would end up with a value of 0. This problem persists when you enable transport delays and whenever the percentage specified in the `+pulse_r/number` option is low enough to enable the pulse to propagate through the module.

To circumvent this problem, when a later event on the input schedules an event on the output that is earlier than the event scheduled by the previous event on the input, VCS cancels both events on the output.

This ensures that the output ends up with the proper value, but what it doesn't do is indicate that something happened on the output between times 15 and 16. You might want to see an error message and an X value pulse on the output indicating there was an undefined event on the output between these simulation times. You see this message and the X value pulse if you include the `+pulse_on_event` compile-time option, specifying pulse on event behavior, as illustrated in [Figure 11-8](#). Pulse on event behavior calls

for an X value pulse on the output after the delay and when there are asymmetrical delays scheduling events on the output that would be canceled by VCS , to output an X value pulse between those events instead.

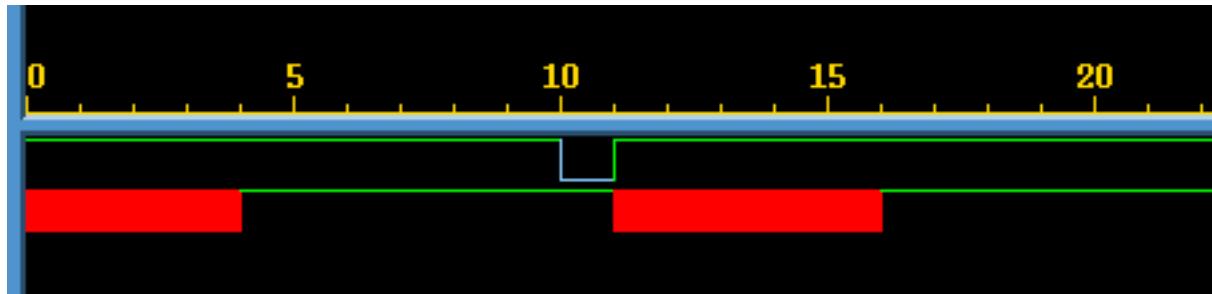
Figure 11-8 Using +pulse_on_event



In most cases where the `+pulse_e/number` and `+pulse_r/number` options already create X value pulses on the output, also including the `+pulse_on_event` option to specify pulse on event behavior will make no change on the output.

Pulse on detect behavior, specified by the `+pulse_on_detect` compile-time option, displays the leading edge of the X value pulse on the output as soon as events on the input, controlled by the `+pulse_e/number` and `+pulse_r/number` options, schedule an X value pulse to appear on the output. Pulse on detect behavior differs from pulse on event behavior in that it calls for the X value pulse to begin before the delay elapses. [Figure 11-9](#) illustrates pulse on detect behavior.

Figure 11-9 Using +pulse_on_detect



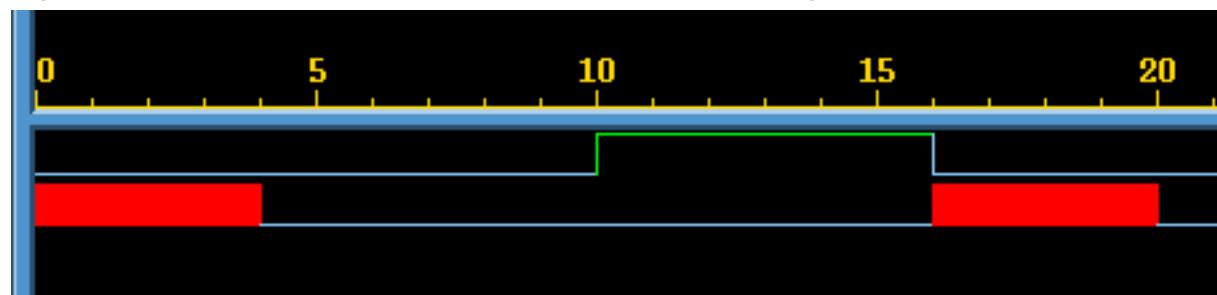
In this example, by including the `+pulse_on_detect` option, VCS causes the leading edge of the X value pulse on the output to begin at time 11 because of an unusual event that occurred on the output between times 15 and 16 because of the rise at simulation time 11.

Using pulse on detect behavior can also show you when VCS has scheduled multiple events for the same simulation time on the output by starting the leading edge of an X value pulse on the output as soon as VCS has scheduled the second event.

For example, a module that models a buffer has a rise time module path delay of 10 time units and a fall time module path delay of 4 time units.

[Figure 11-10](#) shows the waveforms for the input and output port when you include the `+pulse_on_detect` option.

Figure 11-10 Pulse on Detect Behavior Showing Multiple Transitions



In the example illustrated in [Figure 11-10](#) the following occurs:

1. At simulation time 0 the input port transitions to 0 scheduling the first event on the output, a transition to 0 at time 4.
2. At time 4 the output transitions to 0.
3. At time 10 the input transitions to 1 scheduling a transition to 1 on the output at time 20.
4. At time 16 the input toggles to 0 scheduling a second event on the output at time 20, a transition to 0. This event also is the trailing edge of a six time unit wide value 1 pulse so the first event changes to a transition to X. There is more than one event for different value changes on the output at time 20, so VCS begins the leading edge of the X value pulse on the output at this time.
5. At time 20 the output toggles to 0, the second scheduled event at this time.

If you did not include the `+pulse_on_detect` option, or substituted the `+pulse_on_event` option, you would not see the X value pulse on the output between times 16 and 20.

Pulse on detect behavior does not just show you when asymmetrical delays schedule multiple events on the output. Other kinds of events can cause multiple events on the output at the same simulation time, such as different transition times on two input ports and different module path delays from these input ports to the output port. Pulse on detect behavior would show you an X value pulse on the output starting when the second event was scheduled on the output port.

Specifying the Delay Mode

It is possible for a module definition to include module path delay that does not equal the cumulative delay specifications in primitive instances and continuous assignment statements in that path.

[Example 11-1](#) shows such a conflict.

Example 11-1 Conflicting Delay Modes

```
'timescale 1 ns / 1 ns
module design (out,in);
output out;
input in;
wire int1,int2;

assign #4 out=int2;

buf #3 buf2 (int2,int1),
      buf1 (int1,in);

specify
  (in => out) = 7;
endspecify
endmodule
```

In [Example 11-1](#), the module path delay is seven time units, but the delay specifications distributed along that path add up to ten time units.

If you include the `+delay_mode_path` compile-time option, VCS ignores the delay specifications in the primitive instantiation and continuous assignment statements and uses only the module path delay. In [Example 11-1](#), it would use the seven time unit delay for propagating signal values through the module.

If you include the `+delay_mode_distributed` compile-time option, VCS ignores the module path delays and uses the delay in the delay specifications in the primitive instantiation and continuous assignment statements. In [Example 11-1](#), it uses the ten time unit delay for propagating signal values through the module.

There are other modes that you can specify:

- If you include the `+delay_mode_unit` compile-time option, VCS ignores the module path delays and changes the delay specification in all primitive instantiation and continuous assignment statements to the shortest time precision argument of all the `'timescale` compiler directives in the source code. (The default time unit and time precision argument of the `'timescale` compiler directive is 1 s). In [Example 11-1](#) the `'timescale` compiler directive has a precision argument of 1 ns. VCS might use this 1 ns as the delay, but if the module definition is used in a larger design and there is another `'timescale` compiler directive in the source code with a finer precision argument, then VCS uses the finer precision argument.
- If you include the `+delay_mode_zero` compile-time option, VCS changes all delay specifications and module path delays to zero.
- If you include none of the compile-time options described in this section, when, as in [Example 11-1](#), the module path delay does not equal the distributed delays along the path, VCS uses the longer of the two.

Using the Configuration File to Disable Timing

You can use the VCS configuration file to disable module path delays, specify blocks, and timing checks for module instances that you specify as well as all instances of module definitions that you specify. You use the instance, module, and tree statements to do this just as you do for applying Radiant Technology. See “[The Configuration File Syntax](#)” on page 10-6 for details on how to do this. The attribute keywords for timing are as follows:

`noIopath`

Specifies disabling the module path delays in the specified module instances.

`noSpecify`

Specifies disabling the specify blocks in the specified module instances.

`noTiming`

Specifies disabling the timing checks in the specified module instances.

Using the `timopt` Timing Optimizer

The `timopt` timing optimizer can yield large speedups for full-timing gate-level designs. The `timopt` timing optimizer makes its optimizations based on the clock signals and sequential devices that it identifies in the design. `timopt` is particularly useful when you use SDF files because SDF files can't be used with Radiant Technology (+rad).

You enable `timopt` with the `+timopt+clock_period` compile-time option, where the argument is the shortest clock period (or clock cycle) of the clock signals in your design. For example:

```
+timopt+100ns
```

This options specifies that the shortest clock period is 100ns.

`timopt` first displays the number of sequential devices that it finds in the design and the number of these sequential devices to which it might be able to apply optimizations. For example:

```
Total Sequential Elements : 2001
Total Sequential Elements 2001, Optimizable 2001
```

`timopt` then displays the percentage of identified sequential devices to which it can actually apply optimizations followed by messages about the optimization process.

```
TIMOPT optimized 75 percent of the design
Starting TIMOPT Delay optimizations
Done TIMOPT Delay Optimizations
DONE TIMOPT
```

The next step is to simulate the design and see if the optimizations applied by `timopt` produce a satisfactory increase in performance. If you are not satisfied there are additional steps that you can take to get more optimizations from `timopt`.

If `timopt` was able to identify all the clock signals and all the sequential devices with an absolute certainty it simply applies its optimizations. If `timopt` is uncertain about a number of clock signals and sequential devices then you can use the following process to maximize `timopt` optimizations:

1. `timopt` writes a configuration file named `timopt.cfg` in the current directory that lists the signals and sequential devices that it finds questionable.
2. You review and edit this file, validating that the signals in the file are, or are not, clock signals and that the module definitions in it are, or are not, sequential devices. If you do not need to make any changes in the file, go to step 5. If you do make changes, go to step 3.
3. Compile your design again with the `+timopt+clock_period` compile-time option.

`timopt` will make the additional optimizations that it did not make, because it was unsure of the signals and sequential devices in the `timopt.cfg` file that it wrote during the first compilation.

4. Look at the `timopt.cfg` file again:
 - If `timopt` wrote no new entries for potential clock signals or sequential devices, go to step 5.
 - If `timopt` wrote new entries, but you make no changes to the new entries, go to step 5.
 - If you make modifications to the new entries, return to step 3.
5. `timopt` does not need to look for any more clock signals and it can assume that the `timopt.cfg` file correctly specifies clock signal and sequential devices. At this point, it just needs to apply the latest optimizations. Compile your design one more time, including the `+timopt` compile-time option, but without its `+clock_period` argument.

6. You now simulate your design using `timopt` optimizations. `timopt` monitors the simulation and makes its optimizations based on its analysis of the design and information in the `timopt.cfg` file. During simulation, if it finds that its assumptions are incorrect, for example the clock period for a clock signal is incorrect, or there is a port for asynchronous control on a module for a sequential device, `timopt` displays a warning message similar to the following:

```
+ Timopt Warning: for clock testbench.clockgen..clk:  
TimePeriod 50ns      Expected 100ns
```

Editing the `timopt.cfg` File

When editing the `timopt.cfg` file, first edit the potential sequential device entries. Edit the potential clock signal only when you have made no changes to the entries for sequential devices.

Editing Potential Sequential Device Entries

The following is an example of sequential devices that `timopt` was not sure of:

```
// POTENTIAL SEQUENTIAL CELLS  
// flop {jknpn} {,};  
// flop {jknpc} {,};  
// flop {tfnpsc} {,};
```

You can remove the comment marks for the module definitions that are, in fact, model sequential devices and which provide the clock port, clock polarity, and optionally asynchronous ports.

A modified list might look like the following:

```
flop { jknpn } { CP, true} ;  
flop { jknpc } { CP, true, CLN} ;  
flop { tfnpc } { CP, true, CLN} ;
```

In this example, CP is the clock port and the keyword `true` indicates that the sequential device is triggered on the posedge of the clock port and CLN is an asynchronous port.

If you uncomment any of these module definitions, then `timopt` might identify additional clock signals that drive these sequential devices. To enable `timopt` to do this:

1. Remove the clock signal entries from the `timopt.cfg` file.
2. Recompile the design with the same `+timopt+clock_period` compile-time option.

`timopt` will write new clock signal entries in the `timopt.cfg` file.

Editing Clock Signal Entries

The following is an example of the clock signal entries:

```
clock {  
    // test.badClock , // 1  
    test.goodClock // 2000  
} {100ns};
```

These clock signals have a period of `100ns` or longer. This time value comes from the `+clock_period` argument that you added to the `+timopt` compile-time option when you first compiled the design. The entry for the signal `test.badClock` is commented out because it connects to a small percentage of the sequential devices in the design. In this instance, it is only 1 of the 2001 sequential

devices that it identified in the design. The entry for the signal test.goodClock is not commented out because it connects to a large percentage of the sequential devices. In this instance, it is 2000 of the 2001 sequential devices in the design.

If a commented out clock signal is a clock signal that you want `timopt` to use when it optimizes the design in a subsequent compilation, then remove the comment characters preceding the signal's hierarchical name.

Negative Timing Checks

Negative timing checks are either `$setuphold` timing checks with negative setup or hold limits, or `$recrem` timing checks with negative recovery or removal limits.

This following sections describe their purpose, how they work, and how to use them:

- “[The Need for Negative Value Timing Checks](#)”
- “[The `\$setuphold` Timing Check Extended Syntax](#)”
- “[The `\$recrem` Timing Check Syntax](#)”
- “[Enabling Negative Timing Checks](#)”
- “[Checking Conditions](#)”
- “[Toggling the Notifier Register](#)”
- “[SDF Back-annotation to Negative Timing Checks](#)”
- “[How VCS Calculates Delays](#)”

- “Using Multiple Non-overlapping Violation Windows”
-

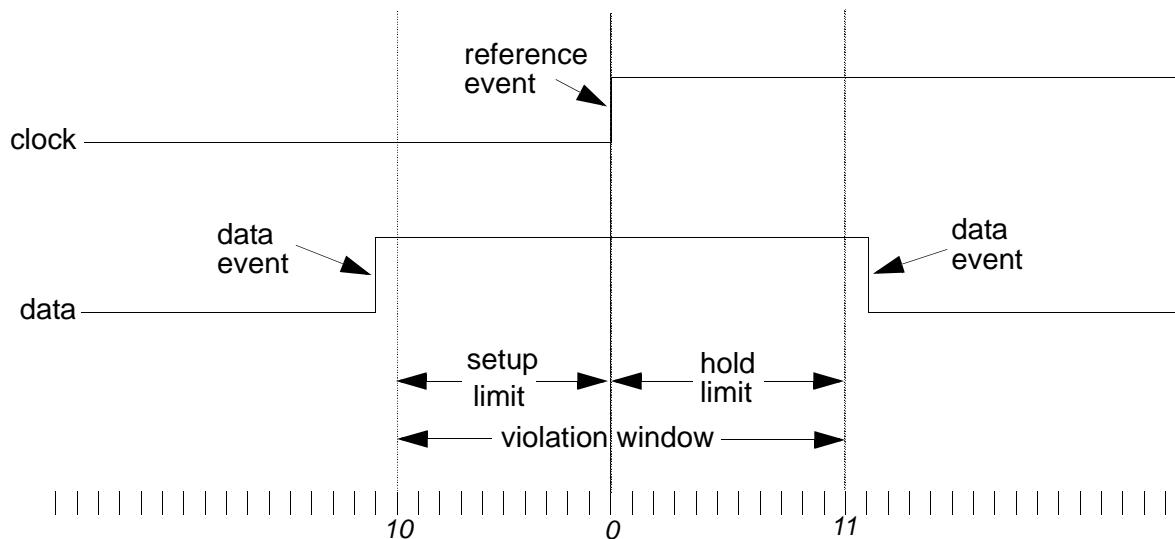
The Need for Negative Value Timing Checks

The `$setuphold` timing check defines a timing violation window of a specified amount of simulation time before and after a reference event, such as a transition on some other signal, for example, a clock signal, in which a data signal must remain constant. A transition on the data signal, called a data event, during the specified window is a timing violation. For example:

```
$setuphold (posedge clock, data, 10, 11, notifyreg);
```

In this example, VCS reports the timing violation if there is a transition on signal `data` less than 10 time units before, or less than 11 time units after, a rising edge on signal `clock`. When there is a timing violation, VCS toggles a notify register, in this example, `notifyreg`. You could use this toggling of a notify register to output an `x` value from a device, such as a sequential flop, when there is a timing violation.

Figure 11-11 Positive Setup and Hold Limits



In this example, both the setup and hold limits have positive values. When this occurs, the violation window straddles the reference event.

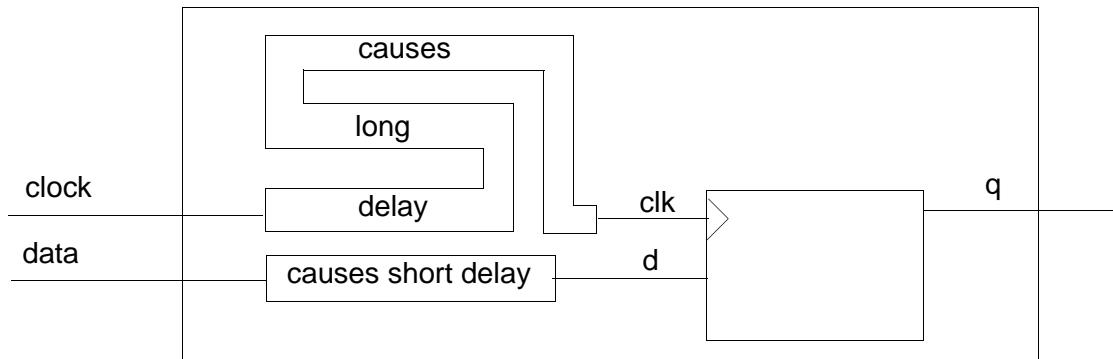
There are cases where the violation window cannot straddle the reference event at the inputs of an ASIC cell. Such a case occurs when:

- The data event takes longer than the reference event to propagate to a sequential device in the cell
- Timing must be accurate at the sequential device
- You need to check for timing violations at the cell boundary

It also occurs when the opposite is true, that is, when the reference event takes longer than the data event to propagate to the sequential device.

When this happens, use the `$setuphold` timing check in the top-level module of the cell to look for timing violations when signal values propagate to that sequential device. In this case, you need to use negative setup or hold limits in the `$setuphold` timing check.

Figure 11-12 ASIC Cell with Long Propagation Delays on Reference Events

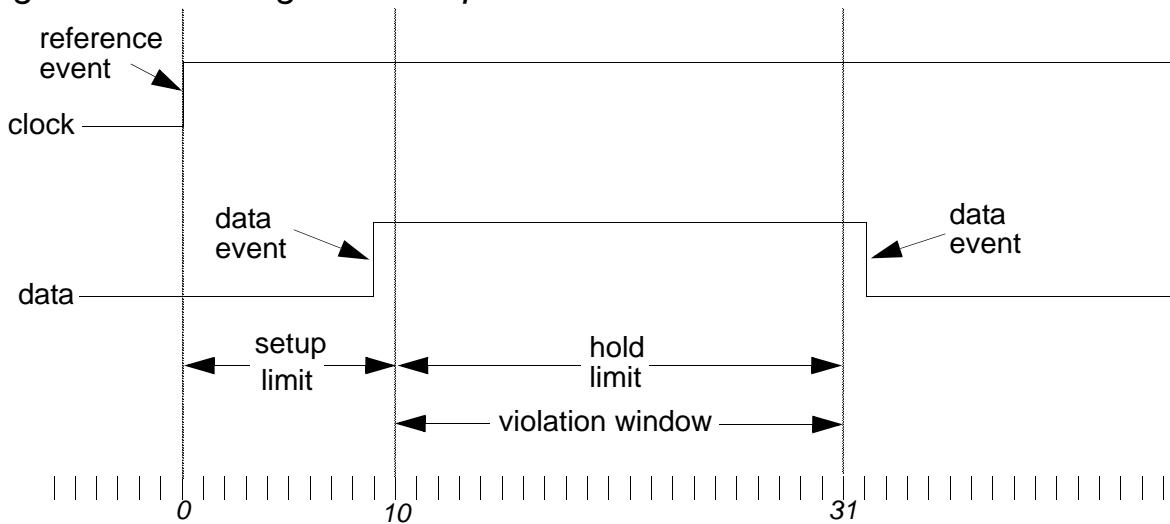


When this occurs, the violation window shifts at the cell boundary so that it no longer straddles the reference event. It shifts to the right when there are longer propagation delays on the reference event. This right shift requires a negative setup limit:

```
$setuphold (posedge clock, data, -10, 31, notifyreg);
```

[Figure 11-13](#) illustrates this scenario.

Figure 11-13 Negative Setup Limit



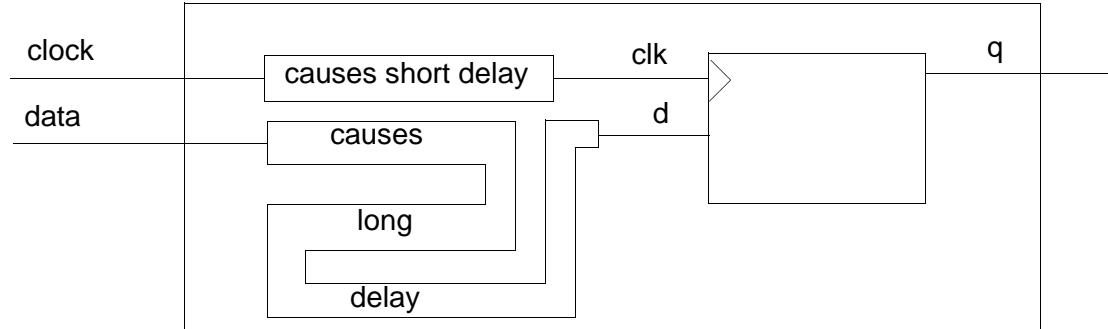
In this example, the `$setupHold` timing check is in the specify block of the top-level module of the cell. It specifies that there is a timing violation if there is a data event between 10 and 31 time units after the reference event on the cell boundary.

This is giving the reference event a “head start” at the cell boundary, anticipating that the delays on the reference event will allow the data events to “catch up” at the sequential device inside the cell.

Note:

When you specify a negative setup limit, its value must be less than the hold limit.

Figure 11-14 ASIC Cell with Long Propagation Delays on Data Events

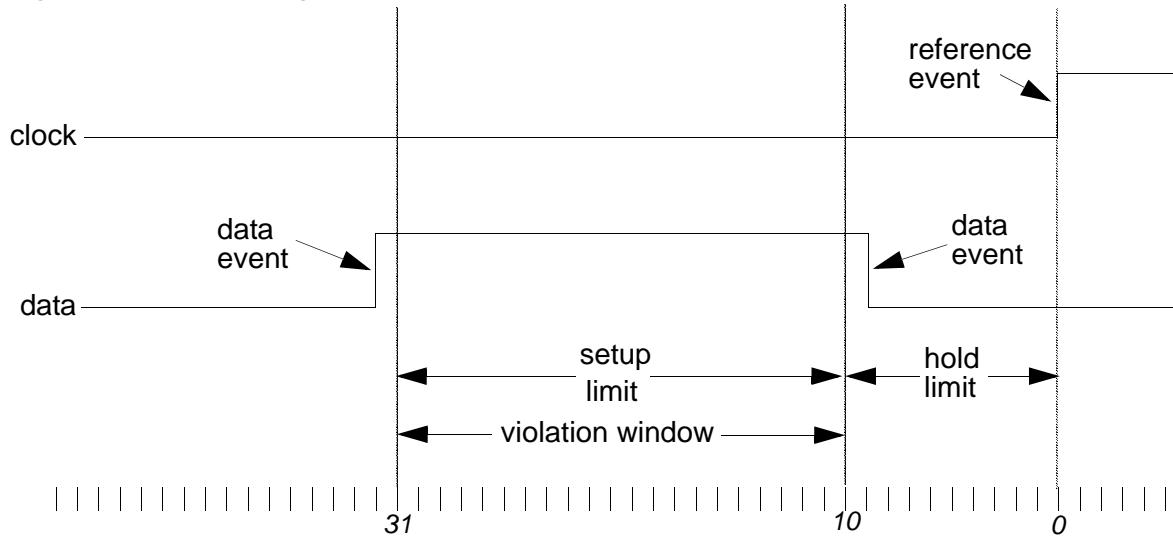


The violation window shifts to the left when there are longer propagation delays on the data event. This left shift requires a negative hold limit:

```
$setuphold (posedge clock, data, 31, -10, notifyreg);
```

[Figure 11-15](#) illustrates this scenario.

Figure 11-15 Negative Hold Limit



In this example, the `$setuphold` timing check is in the specify block of the top-level module of the cell. It specifies that there is a timing violation if there is a data event between 31 and 10 time units before the reference event on the cell boundary.

This is giving the data events a “head start” at the cell boundary, anticipating that the delays on the data events will allow the reference event to “catch up” at the sequential device inside the cell.

Note:

When you specify a negative hold limit, its value must be less than the setup limit.

To implement negative timing checks, VCS creates delayed versions of the signals that carry the reference and data events and an alternative violation window where the window straddles the delayed reference event.

You can specify the names of the delayed versions by using the extended syntax of the `$setuphold` system task, or by allowing VCS to name them internally.

The extended syntax also allows you to specify expressions for additional conditions that must be true for a timing violation to occur.

The `$setuphold` Timing Check Extended Syntax

The `$setuphold` timing check has the following extended syntax:

```
$setuphold(reference_event, data_event, setup_limit,  
hold_limit, notifier, [timestamp_cond, timecheck_cond,  
delayed_reference_signal, delayed_data_signal]);
```

The following additional arguments are optional:

`timestamp_cond`

This argument specifies the condition which determines whether or not VCS reports a timing violation.

In the setup phase of a `$setuphold` timing check, VCS records or “stamps” the time of a data event internally so that when a reference event occurs, it can compare the times of these events to see if there is a setup timing violation. If the condition specified by this argument is false, VCS does not record or “stamp” the data event so there cannot be a setup timing violation.

Similarly, in the hold phase of a `$setuphold` timing check, VCS records or “stamps” the time of a reference event internally so that when a data event occurs, it can compare the times of these events to see if there is a hold timing violation. If the condition specified by this argument is false, VCS does not record or “stamp” the reference event so there cannot be a hold timing violation.

`timecheck_cond`

This argument specifies the condition which determines whether or not VCS reports a timing violation.

In the setup phase of a `$setuphold` timing check, VCS compares or “checks” the time of the reference event with the time of the data event to see if there is a setup timing violation. If the condition specified by this argument is false, VCS does not make this comparison and so there is no setup timing violation.

Similarly, in the hold phase of a `$setuphold` timing check, VCS compares or “checks” the time of a data event with the time of a reference event to see if there is a hold timing violation. If the condition specified by this argument is false, VCS does not make this comparison and so there is no hold timing violation.

`delayed_reference_signal`

The name of the delayed version of the reference signal.

`delayed_data_signal`

The name of the delayed version of the data signal.

The following example demonstrates how to use the extended syntax:

```
$setuphold(ref, data, -4, 10, notifrl, stampreg==1, , d_ref,  
          d_data);
```

In this example, the `timestamp_cond` argument specifies that reg `stampreg` must equal `1` for VCS to “stamp” or record the times of data events in the setup phase or “stamp” the times of reference events in the hold phase. If this condition is not met, and stamping

does not occur, VCS will not find timing violations no matter what the time is for these events. Also in the example, the delayed versions of the reference and data signals are named `d_ref` and `d_data`.

You can use these delayed signal versions of the signals to drive sequential devices in your cell model. For example:

```

module DFF(D,RST,CLK,Q);
  input D,RST,CLK;
  output Q;
  reg notifier;
  DFF_UDP d2(Q,dCLK,dD,dRST,notifier);
  specify
    (D => Q) = 20;
    (CLK => Q) = 20;
    $setuphold(posedge CLK,D,-5,10,notifier,,,dCLK,dD);
    $setuphold(posedge CLK,RST,-8,12,notifier,,,dCLK,
               dRST);
  endspecify
endmodule

primitive DFF_UDP(q,clk,data,rst,notifier);
  output q; reg q;
  input data,clk,rst,notifier;

table
// clock  data rst   notifier  state  q
// -----
r    0     0      ?       : ? : 0 ;
r    1     0      ?       : ? : 1 ;
f    ?     0      ?       : ? : - ;
?    ?     r      ?       : ? : 0 ;
?    *     ?      ?       : ? : - ;
?    ?     ?      *       : ? : x ;
endtable
endprimitive

```

In this example, the DFF_UDP user-defined primitive is driven by the delayed signals dClk, dD, dRST, and the notifier reg.

Negative Timing Checks for Asynchronous Controls

The \$recrem timing check is used for checking how close asynchronous control signal transitions are to clock signals. Similar to the setup and hold limits in \$setuphold timing checks, the \$recrem timing check has recovery and removal limits. The recovery limit specifies how much time must elapse after a control signal toggles from its active state before there is an active clock edge. The removal limit specifies how much time must elapse after an active clock edge before the control signal can toggle from its active state.

In the same way a reference signal, such as a clock signal and data signal can have different propagation delays from the cell boundary to a sequential device inside the cell, there can be different propagation delays between the clock signal and the control signal. For this reason, there can be negative recovery and removal limits in the \$recrem timing check.

The \$recrem Timing Check Syntax

The \$recrem timing check syntax is very similar to the extended syntax for \$setuphold:

```
$recrem(reference_event, data_event, recovery_limit,  
removal_limit, notifier, [timestamp_cond, timecheck_cond,  
delayed_reference_signal, delayed_data_signal]);
```

`reference_event`

Typically the reference event is the active edge on a control signal, such as a clear signal. Specify the active edge with the `posedge` or `negedge` keyword.

`data_event`

Typically, the data event occurs on a clock signal. Specify the active edge on this signal with the `posedge` or `negedge` keyword.

`recovery_limit`

Specifies how much time must elapse after a control signal, such as a clear signal toggles from its active state (the reference event), before there is an active clock edge (the data event).

`removal_limit`

Specifies how much time must elapse after an active clock edge (the data event), before the control signal can toggle from its active state (the reference event).

`notifier`

A register whose value VCS toggles when there is a timing violation.

`timestamp_cond`

This argument specifies the condition which determines whether or not VCS reports a timing violation.

In the recovery phase of a `$recrem` timing check, VCS records or “stamps” the time of a reference event internally so that when a data event occurs it can compare the times of these events to see if there is a recovery timing violation. If the condition specified by this argument is false, VCS does not record or “stamp” the reference event so there cannot be a recovery timing violation.

Similarly, in the removal phase of a `$recrem` timing check, VCS records or “stamps” the time of a data event internally so that when a reference event occurs, it can compare the times of these events to see if there is a removal timing violation. If the condition specified by this argument is false, VCS does not record or “stamp” the data event so there cannot be a removal timing violation.

`timecheck_cond`

This argument specifies the condition which determines whether or not VCS reports a timing violation.

In the recovery phase of a `$recrm` timing check, VCS compares or “checks” the time of the data event with the time of the reference event to see if there is a recovery timing violation. If the condition specified by this argument is false, VCS does not make this comparison and so there is no recovery timing violation.

Similarly, in the removal phase of a `$recrm` timing check, VCS compares or “checks” the time of a reference event with the time of a data event to see if there is a removal timing violation. If the condition specified by this argument is false, VCS does not make this comparison and so there is no removal timing violation.

`delayed_reference_signal`

The name of the delayed version of the reference signal, typically a control signal.

`delayed_data_signal`

The name of the delayed version of the data signal, typically a clock signal.

Enabling Negative Timing Checks

To use a negative timing check you must include the `+neg_tchk` compile-time option when you compile your design. If you omit this option, VCS changes all negative limits to 0.

If you include the `+no_notifier` compile-time option with the `+neg_tchk` option, you only disable notifier toggling. VCS still creates the delayed versions of the reference and data signals and displays timing violation messages.

Conversely, if you include the `+no_tchk_msg` compile-time option with the `+neg_tchk` option, you only disable timing violation messages. VCS still creates the delayed versions of the reference and data signals and toggles notifier regs when there are timing violations.

If you include the `+neg_tchk` compile-time option but also include the `+notimingcheck` or `+nospecify` compile-time options, VCS does not compile the `$setuphold` and `$recrem` timing checks into the `simv` executable. However, it does create the signals that you specified in the `delayed_reference_signal` and `delayed_data_signal` arguments, and you can use these to drive sequential devices in the cell. Note that there is no delay on these "delayed" arguments and they have the same transition times as the signals specified in the `reference_event` and `data_event` arguments.

Similarly, if you include the `+neg_tchk` compile-time option and then include the `+notimingcheck` runtime option instead of the compile-time option, you disable the `$setuphold` and `$recrem` timing checks that VCS compiled into the executable. At compile time, VCS creates the signals that you specified in the `delayed_reference_signal` and `delayed_data_signal` arguments, and you can use them to drive sequential devices in the cell, but the `+notimingcheck` runtime option disables the delay on these "delayed" versions.

Other Timing Checks Using the Delayed Signals

When you enable negative timing limits in the `$setuphold` and `$recrm` timing checks, and have VCS create delayed versions of the data and reference signals, by default the other timing checks also use the delayed versions of these signals. You can prevent the other timing checks from doing this with the `+old_ntc` compile-time option.

Having the other timing checks use the delayed versions of these signals is particularly useful when the other timing checks use a notifier register to change the output of the sequential element to `x`.

Example 11-2 Notifier Register Example for Delayed Reference and Data Signals

```
`timescale 1ns/1ns

module top;
    reg clk, d;
    reg rst;
    wire q;

    dff dff1(q, clk, d, rst);

    initial begin
        $monitor($time,,clk,,d,,q);
        rst = 0; clk = 0; d = 0;
        #100 clk = 1;
        #100 clk = 0;
        #10 d = 1;
        #90 clk = 1;
        #1 clk = 0; // width violation
        #100 $finish;
    end
endmodule

module dff(q, clk, d, rst);
    output q;
    input clk, d, rst;
    reg notif;

    DFF_UDP(q, d_clk, d_d, d_rst, notif);

    specify
        $setuphold(posedge clk, d, -10, 20, notif, , , d_clk,
                   d_d);
        $setuphold(posedge clk, rst, 10, 10, notif, , , d_clk,
                   d_rst);
        $width(posedge clk, 5, 0, notif);
    endspecify
endmodule
```

```

primitive DFF_UDP(q,data,clk,rst,notifier);
output q; reg q;
input data,clk,rst,notifier;

table
// clock  data  rst   notifier  state  q
// -----
    r      0      0      ?      :      ?      :      0 ;
    r      1      0      ?      :      ?      :      1 ;
    f      ?      0      ?      :      ?      :      - ;
    ?      ?      r      ?      :      ?      :      0 ;
    ?      *      ?      ?      :      ?      :      - ;
    ?      ?      ?      *      :      ?      :      x ;
endtable
endprimitive

```

In this example, if you include the `+neg_tchk` compile-time option, the `$width` timing check uses the delayed version of signal `clk`, named `d_clk`, and the following sequence of events occurs:

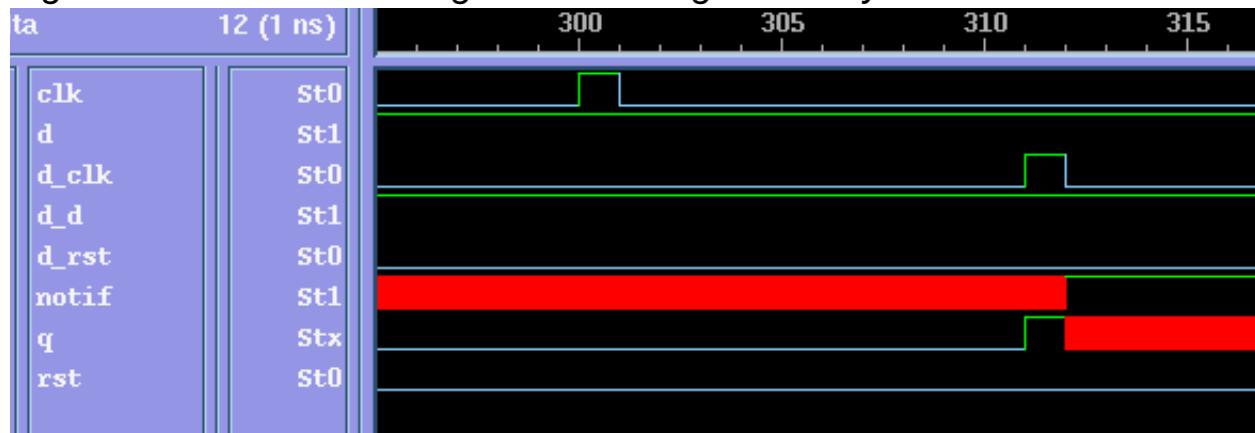
1. At time 311, the delayed version of the clock transitions to 1, causing output `q` to toggle to 1.
2. At time 312, the narrow pulse on the clock causes a width violation:

```
"test1.v", 31: Timing violation in top.dff1
$width( posedge clk:300,    : 301, limit: 5);
```

The timing violation message looks like it occurs at time 301, but you do not see it until time 312.

3. Also at time 312, `reg notif` toggles from `x` to 1. This changes output `q` from 1 to `x`. There are no subsequent changes on output `q`.

Figure 11-16 Other Timing Checks Using the Delayed Versions

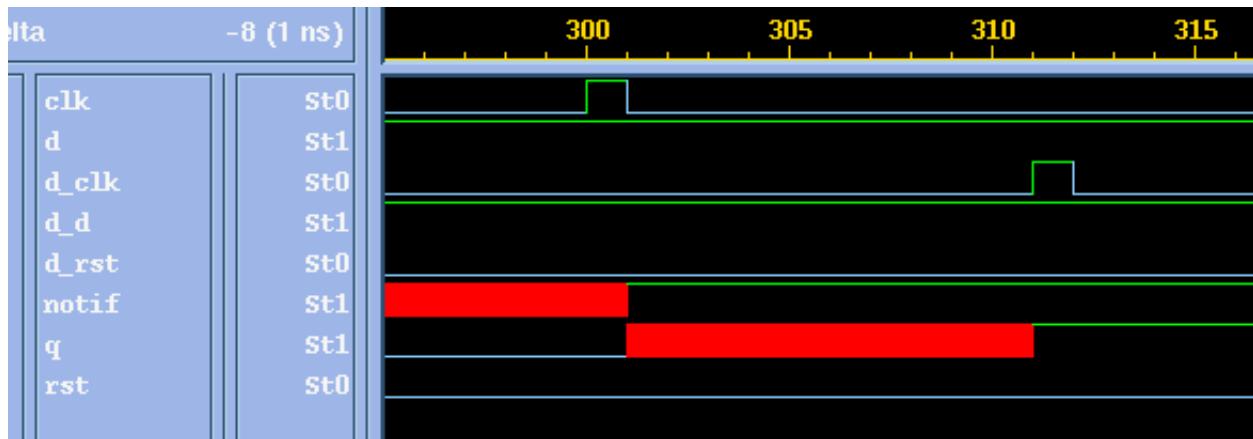


If you include both the `+neg_tchk` and `+old_ntc` compile-time options, the `$width` timing check does not use the delayed version of signal `clk`, causing the following sequence of events to occur:

1. At time 301, the narrow pulse on signal `clk` causes a width violation:


```
"test1.v", 31: Timing violation in top.dff1
$width( posedge clk:300,    : 301, limit: 5);
```
2. Also at time 301, the notifier reg named `notif` toggles from `x` to `1`. In turn, this changes the output `q` of the user-defined primitive `DFF_UDP` and module instance `dff1` from `0` to `x`.
3. At time 311, the delayed version of signal `clk`, named `d_clk`, reaches the user-defined primitive `DFF_UDP`, thereby changing the output `q` to `1`, erasing the `x` value on this output.

Figure 11-17 Other Timing Checks Not Using the Delayed Versions



The timing violation, as represented by the `x` value, is lost to the design. If a module path delay that is greater than ten time units was used for the module instance, the `x` value would not appear on the output at all.

For this reason, Synopsys does not recommend using the `+old_ntc` compile-time option. It exists only for unforeseen circumstances.

Checking Conditions

VCS evaluates the expressions in the `timestamp_cond` and `timecheck_cond` arguments either when there is a value change on the original reference and data signals at the cell boundary, or when the value changes propagate from the delayed versions of these signals at the sequential device inside the cell. It decides when to evaluate the expressions depending on which signals are the operands in these expressions. Note the following:

- If the operands in these expressions are neither the original nor the delayed versions of the reference or data signals, and if these operands are signals that do not change value between value changes on the original reference and data signals and their delayed versions, then it does not matter when VCS evaluates these expressions.
- If the operands in these expressions are delayed versions of the original reference and data signals, then you want VCS to evaluate these expressions when there are value changes on the delayed versions of the reference and data signals. VCS does this by default.
- If the operands in these expressions are the original reference and data signals and not the delayed versions, then you want VCS to evaluate these expressions when there are value changes on the original reference and data signals. To specify evaluating these expressions when the original reference and data signals change value, include the `+NTC2` compile-time option.

Toggling the Notifier Register

VCS waits for a timing violation to occur on the delayed versions of the reference and data signals before toggling the notifier register. Toggling means the following value changes:

- X to 0
- 0 to 1
- 1 to 0

VCS does not change the value of the notifier register if you have assigned a Z value to it.

SDF Back-annotation to Negative Timing Checks

You can back-annotate negative setup and hold limits from SDF files to `$setuphold` timing checks and negative recovery and removal limits from SDF files to `$recrem` timing checks, if the following conditions are met:

- You included the arguments for the names of the delayed reference and data signals in the timing checks.
- You compiled your design with the `+neg_tchk` compile-time option.
- For all `$setuphold` timing checks, the positive setup or hold limit is greater than the negative setup or hold limit.
- For all `$recrem` timing checks, the positive recovery or removal limit is greater than the negative recovery or removal limit.

As documented in the OVI SDF3.0 specification:

- TIMINGCHECK statements in the SDF file back-annotate timing checks in the model which match the edge and condition arguments in the SDF statement.
- If the SDF statement specifies SCOND or CCOND expressions, they must match the corresponding `timestamp_cond` or `timecheck_cond` in the timing check declaration for back-annotation to occur.
- If there is no SCOND or CCOND expressions in the SDF statement, all timing checks that otherwise match are back-annotated.

How VCS Calculates Delays

This section describes how VCS calculates the delays of the delayed versions of reference and data signals. It does not describe how you use negative timing checks; it is supplemental material intended for users who would like to read more about how negative timing checks work in VCS.

VCS uses the limits you specify in the `$setuphold` or `$recrem` timing check to calculate the delays on the delayed versions of the reference and data signals. For example:

```
$setuphold(posedge clock,data,-10,20, , , , del_clock,  
           del_data);
```

This specifies that the propagation delays on the reference event (a rising edge on signal clock), are more than 10 but less than 20 time units more than the propagation delays on the data event (any transition on signal data).

So when VCS creates the delayed signals, `del_clock` and `del_data`, and the alternative violation window that straddles a rising edge on `del_clock`, VCS uses the following relationship:

$$20 > (\text{delay on } \text{del_clock} - \text{delay on } \text{del_data}) > 10$$

There is no reason to make the delays on either of these delayed signals any longer than they have to be so the delay on `del_data` is 0 and the delay on `del_clock` is 11. Any delay on `del_clock` between 11 and 19 time units would report a timing violation for the `$setuphold` timing check.

Multiple timing checks, that share reference or data events, and specified delayed signal names, can define a set of delay relationships. For example:

```
$setuphold(posedge CP,D,-10,20, notifier, , ,
           del_CP, del_D);
$setuphold(posedge CP,TI,20,-10, notifier, , ,
           del_CP, del_TI);
$setuphold(posedge CP,TE,-4,8,  notifier, , ,
           del_CP, del_TE);
```

In this example:

- The first `$setuphold` timing check specifies the delay on `del_CP` is more than 10 but less than 20 time units more than the delay on `del_D`.
- The second `$setuphold` timing check specifies the delay on `del_TI` is more than 10 but less than 20 time units more than the delay on `del_CP`.
- The third `$setuphold` timing check specifies the delay on `del_CP` is more than 4 but less than 8 time units more than the delay on `del_TE`.

Therefore:

- The delay on `del_D` is 0 because its delay does not have to be more than any other delayed signal.
- The delay on `del_CP` is 11 because it must be more than 10 time units more than the 0 delay on `del_D`.

- The delay on `del_TE` is 4 because the delay on `del_CP` is 11. The 11 makes the possible delay on `del_TE` larger than 3, but less than 7. The delay cannot be 3 or less, because the delay on `del_CP` is less than 8 time units more than the delay on `del_TE`. VCS makes the delay 4 because it always uses the shortest possible delay.
- The delay on `del_TI` is 22 because it must be more than 10 time units more than the 11 delay on `del_CP`.

In unusual and rare circumstances, multiple `$setuphold` and `$recrem` timing checks, including those that have no negative limits, can make the delays on the delayed versions of these signals mutually exclusive. When this happens, VCS repeats the following procedure until the signals are no longer mutually exclusive:

1. Sets one negative limit to 0.
2. Recalculates the delays of the delayed signals.

Using Multiple Non-overlapping Violation Windows

The `+overlap` compile-time option enables accurate simulation of multiple violation windows for the same two signals when the following conditions occur:

- The violation windows are specified with negative delay values that are back-annotated from an SDF file.
- The violation windows do not converge or overlap.

When these conditions are met, the default behavior of VCS is to replace the negative delay values with zeros so that the violation windows overlap. Consider the following code example:

```
'timescale 1ns/1ns
module top;
reg in1, clk;
wire out1;

FD1  fd1_1 ( .d(in1), .cp(clk), .q(out1) ) ;

initial
begin
    $sdf_annotate("overlap1.sdf");
in1 = 0;
#45 in1=1;
end

initial
begin
    clk=0;
#50 clk = 1;
#50 clk = 0;
end
endmodule

module FD1 (d, cp, q);
input d, cp;
output q;
wire q;
reg notifier;
reg q_reg;

always @(posedge cp)
q_reg = d;

assign q = q_reg;

specify
```

```

$setuphold( posedge cp, negedge d, 40, 30, notifier);
$setuphold( posedge cp, posedge d, 20, 10, notifier);
endspecify
endmodule

```

The SDF file contains the following to back-annotate negative delay values:

```

(CELL
  (CELLTYPE "FD1")
  (INSTANCE top.fd1_1)
  (TIMINGCHECK
    (SETUPHOLD (negedge d) (posedge cp) (40) (-30))
    (SETUPHOLD (posedge d) (posedge cp) (20) (-10))
  )
)

```

So the timing checks are now:

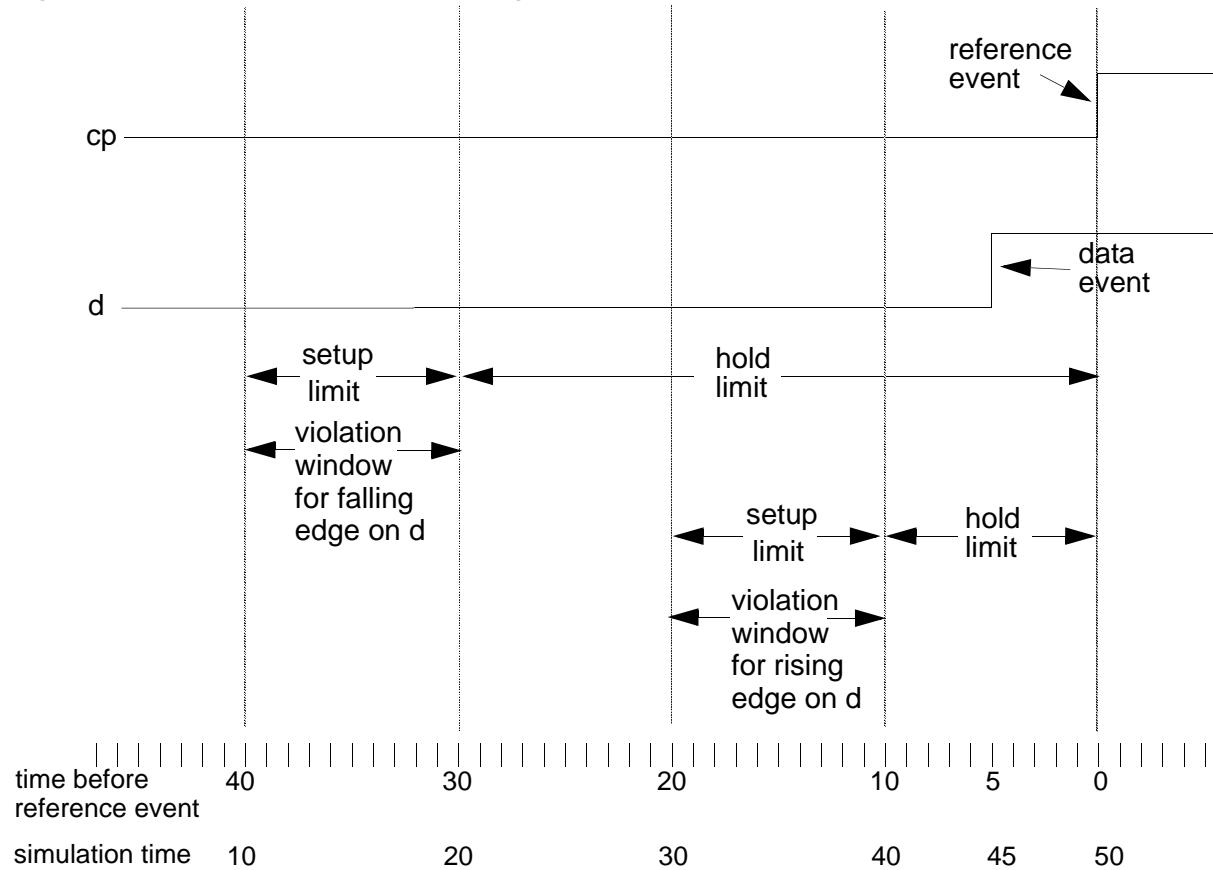
```

$setuphold( posedge cp, negedge d, 40, -30, notifier);
$setuphold( posedge cp, posedge d, 20, -10, notifier);

```

The violation windows and the transitions that occur on signals `top.fd1_1.cp` and `top.fd1_1.d` are shown in [Figure 11-18](#).

Figure 11-18 Non-Overlapping Violation Windows



The `$setuphold` timing checks now specify:

- A violation window for a falling edge on signal *d* between 40 and 30 time units before a rising edge on signal *cp*
- A violation window for a rising edge on signal *d* between 20 and 10 time units before a rising edge on signal *cp*

The testbench module `top` applies stimulus so that the following transitions occur:

1. A rising edge on signal *d* at time 45
2. A rising edge on signal *cp* at time 50

The rising edge on signal d at time 45 is not inside the violation window for a rising edge on signal d. If you include the +overlap compile-time option, you will not see a timing violation. This behavior is desired because there is no transition in the violation windows so VCS should not display a timing violation.

The +overlap option tells VCS not to change the violation windows, just like it would if the windows overlapped.

If you omit the +overlap option, VCS does what Verilog simulators traditionally do, which is both pessimistic and inaccurate:

1. During compilation, VCS replaces the -30 and -10 negative delay values in the \$setuphold timing checks with 0 values. It displays the following warning:

```
Warning: Negative Timing Check delays did not converge,  
Setting minimum constraint to zero and using approximation  
solution (  
"sourcefile", line_number_of_second_timing_check)
```

VCS alters the violation windows:

- For the falling edge, the window starts 40 time units before the reference event and ends at the reference event.
- For the rising edge, the window starts 20 time units before the reference event and also ends at the reference event.

VCS alters the windows so that they overlap or converge.

2. During simulation, at time 50 (reference event), VCS displays the timing violation message:

```
"sourcefile.v", line_number_of_second_timing_check:  
Timing violation in top.fd1_1  
$setuphold( posedge cp:50 posedge d:45, limits (20,0)
```

) ;

The rising edge on signal d is in the altered violation window for a rising edge on d that starts 20 time units *before* the reference event and now ends *at* the reference event. The rising edge on signal d occurs five time units before the reference event.

12

Coverage

This chapter contains the following sections:

- “Code Coverage”
- “Unified Report Generator (URG)”

Using VCS you can generate the following two types of coverage database:

- Code Coverage

Code coverage tells you how well you have exercised your implementation without regard for the verification intent. It provides you with information on the lines, paths, and branches of your design that were executed during simulation. It also monitors the signal values, and transitions during simulation, and reports them as toggle and FSM coverage, respectively.

- Functional Coverage

Functional coverage tells you how well you have exercised your verification plan without consideration of your implementation. It provides you with information of your testbench variable and signal values and their state transitions. You can enable cross-coverage between variables and signals. See, “[Functional Coverage](#)” on page 14-4 and “[Functional Coverage](#)” on page 13-30.

VCS writes both the code and functional coverage database during simulation. Later, you can use the “[Unified Report Generator \(URG\)](#)” to generate HTML or text coverage reports.

This chapter is a high-level introduction to the available coverage metrics, and the URG for report generation. For more information on coverage metrics, see the *VCS/VCS MX Coverage Metrics User Guide*.

Code Coverage

VCS provides you the following types of code coverage metrics:

- “[Line Coverage](#)”
- “[Condition Coverage](#)”
- “[Toggle Coverage](#)”
- “[Finite State Machine \(FSM\) Coverage](#)”
- “[Branch Coverage \(for Verilog Only Designs\)](#)”

You can enable code coverage using the coverage option `-cm line|cond|fsm|tgl|path|branch` during both compilation and simulation. During compilation, VCS enables the specified coverage metric, and monitors and writes the database for the design during simulation.

Note:

If you omit the coverage options at runtime, no coverage information is saved. However, your simulation will run faster, although not as fast as if you compiled without coverage.

By default, VCS creates the coverage database within a directory called `simv.cm`, based on the binary executable name, `simv`. However, you can also use the `-cm_dir` option to point to any directory where you want VCS to write the coverage information. If the specified directory does not exist, VCS creates the directory in the specified path, and writes the coverage information.

Note:

If you compile the design with the `-cm_dir` option, and then move `simv.cm`, you must use `-cm_dir` at runtime to point to the new location of `simv.cm`.

Line Coverage

Line coverage is the most basic metric, and is typically run on behavioral RTL simulations. It provides you with information on which lines of the code were or were not executed during simulation.

In line coverage, VCS keeps track of the following in the source code:

- Individual procedural statements

- Procedural statement blocks
- Procedural statement block type
- Missing (implied) conditional statements
- Branches for conditional statements

You enable line coverage by using the `-cm line` option during both compilation and simulation as shown in the “[Usage Model](#)” later in this chapter.

Condition Coverage

Condition coverage monitors the conditional expressions in your code. It tells you because of which vector the expression is true and therefore covered. For example:

```
out <= b || c;
```

In this expression, `out` is true only when either `b` and `c` are true. `out` is covered only when both `b` and `c` have caused it to be true.

Conditions are expressions and subexpressions that control the execution of code or the assignment of values to signals. By default, conditions are the following:

- The conditional expression used with the `? :` conditional operator.
- The subexpressions that are the operands of the logical `and` `&&` and logical `or` `||` operators in the conditional expression used with the `? :` conditional operator.
- The subexpressions that are the operands of the logical `and` `&&` and `or` `||` operators in the conditional expression in an `if` statement.

- The subexpressions that are the operands of the logical `and` `&&` and `or` `||` operators in the continuous and procedural assignment statements.

Because conditions contribute to the execution of the corresponding block of code, it is important to exercise all conditions so that the conditional expressions can be validated for correctness.

You can enable condition coverage by using the `-cm cond` option during both compilation and simulation as shown in the “[Usage Model](#)” later in this chapter.

Condition coverage, then, reveals how subexpressions in statements are evaluated during the simulation. By default, only the subexpressions of the logical operators `&&` and `||` and the conditional operand of the conditional or ternary operator `? :` are conditions for condition coverage. However, you can also add the following conditions:

- The subexpressions of most other operators.
- The signals in event controls that control the execution of an entire always block (the sensitivity list). If you use an implicit event control expression list, and specify that the signals in event controls are conditions, VCS monitors the signal implied by this implicit event control expression list for condition coverage.

For more information, see the *VCS/VCS MX Coverage Metrics User Guide*.

Toggle Coverage

Toggle coverage monitors each net and register for any value transition from 0 to 1 and 1 to 0. This is typically useful for gate-level simulations. VCS generates metrics for total coverage of nets and registers, which provides you with the amount of activity occurring on all the elements, therefore, giving you a clear indication of how much testing is actually being performed at the gate level.

Missing transitions of values provide definitive conclusions about inactive elements and unexercised portions of the design, and the statistics produced for each module can be examined to quickly determine areas of low coverage. This helps you to write tests to address the missing activity.

You can enable toggle coverage by using the `-cm tgl` option during both compilation and simulation as shown in the “[Usage Model](#)” later in this chapter.

For more information, see the *VCS/VCS MX Coverage Metrics User Guide*.

Finite State Machine (FSM) Coverage

FSM coverage monitors the states, state transitions, and sequence of states executed in the design. In higher level Verilog, a group of statements can be a higher level of abstraction of an FSM. This information is typically obtained in behavioral or RTL simulations. VCS treats a group of statements as an FSM, if the group contains a procedural assignment statement to assign the current state of the FSM. The values assigned in the group of statements must be

clearly identifiable as constants and there must be a dependency between the current state and the next state. The following group of Verilog statements is an FSM:

```
module fsm(clk, rstn);
    input clk, rstn;
    reg [3:0] current,next;

    always @(posedge clk or negedge rstn) begin
        if (~rstn)
            current <= 4'b0001;
        else
            current <= next;
    end
    always @ (current) begin
        case (1'b)
            current[0]: next = 4'b0010;
            current[1]: next = 4'b0100;
            current[2]: next = 4'b1000;
            current[3]: next = 4'b0001;
        endcase
    end
endmodule
```

When you write such a group of statements, you want to know if the statements made the assignments to all possible states of the FSM and all possible transitions between states.

If coded, VCS can automatically extract the FSM from your design. This means that VCS can identify the group of statements as an FSM, keep track of the current states of the FSM and report or display to you state and state transition coverage information using cmView.

Note:

VCS does not automatically extract an FSM where the current state is stored in a scalar register or variable, such as integer etc.

You can enable FSM coverage by using the `-cm fsm` option during both compilation and simulation as shown in the “[Usage Model](#)” later in this chapter.

For more information, see the [VCS/VCS MX Coverage Metrics User Guide](#).

Branch Coverage (for Verilog Only Designs)

Branch coverage analyzes how `if` and `case` statements and the ternary operator (`? :`) establish branches of execution in your Verilog design. It shows you vectors of signal or expression values that enable or prevent simulation events.

Consider the code in [Example 12-1](#).

Example 12-1 Branch Coverage Code Example

```
case (r1)
  1'b1    : if (r2 && r3)
              r4 = (r5 && r6) ? 1'b0 : 1'b1;
  1'b0    : if (r7 && r8)
              r9 = (r10 && r11) ? 1'b0 : 1'b1;
  default : $display("no op");
endcase
```

In this block of code, there are procedural assignment statements where the value assigned is controlled by the ternary operator (`? :`). These in turn are controlled by `if` statements, and the `if` statements are controlled by a `case` statement.

In this block of code, the possible vectors of signal or expression values that result in simulation events or prevent simulation events, are as follows:

```
r1      (r2 && r3)  (r5 && r6)  (r7 && r8)  (r10 && r11)

1          1          1          -          -
1          1          0          -          -
1          0          -          -          -
0          -          -          1          1
0          -          -          1          0
0          -          -          0          -
default    -          -          -          -
```

In the first vector, for example, `r1` is `1'b1` and the expression `(r2 && r3)` is true, so the value of `r4` depends on the value of `(r5 && r6)`. The values of `(r7 && r8)` and `(r10 && r11)` do not matter.

In the third vector, for another example, `r1` is `1'b1` but the expression `(r2 && r3)` is false. The values of the other expressions don't matter, and nothing happens.

Branch coverage shows you these vectors and then tells you whether or not these vectors ever occurred during simulation - that is to say, whether or not they were covered.

By default, VCS does not monitor for branch coverage `if` and `case` statements and use of the ternary operator `(? :)` if they are in user-defined tasks or functions, or in code that executes as a result of a `for` loop. You can, however, enable branch coverage in this code. For more information, refer to the *VCS/VCS MX Coverage Metrics User Guide*.

Usage Model

The usage model to invoke the code coverage is shown below:

Compilation

```
% vcs [cover_options] [compile_options] file1.v file2.v
```

Simulation

```
% simv [cover_options] [run_options]
```

Generating Reports

```
% urg [urg_options] -dir [coverage_dbs]
```

For additional information regarding report generation, see the section entitled “[Unified Report Generator \(URG\)](#)” later in this chapter.

In the above usage model, *compile_options* are compilation options. *cover_options* are coverage related compilation options. For example, -cm line and -cm line+tgl. For additional information, see “[Options for Compiling For Coverage Metrics](#)” on [page B-19](#).

The following section describes some of the commonly used coverage related compilation options.

Compilation and Runtime Options

This section describes compilation and runtime options to perform the following tasks:

- Specifying more than one type of coverage

Use the "+" character to specify more than one type of coverage during compilation and runtime. For example, the following command line will monitor and write coverage database for line, condition and toggle coverage:

```
% vcs -cm line+cond+tgl top
% simv -cm line+cond+tgl
```

- Specifying a path for the coverage directory

By default, VCS writes coverage information of a design in the `simv.cm` directory. However, you can use the `-cm_dir directory_path` option during both compilation to specify a directory where VCS can write the coverage information.

Note:

If you compile the design with the `-cm_dir` option, and then move `simv.cm`, you must use `-cm_dir` at runtime to point to the new location of `simv.cm`.

- Using a configuration file to control coverage and reporting

The `-cm_hier` compile-time option takes a configuration file argument. You can use this configuration file to specify the parts of the design that you want VCS to:

- Compile exclusively for coverage

- Exclude from coverage

The `-cm_hier` option can also be used on the `vcs -cm_pp` command line to control the scope of the reports.

For more information on the configuration file, refer to the VCS/
VCS MX Coverage Metrics User Guide.

- Controlling FSM coverage using a configuration file

The `-cm_fsmcfg` option takes a configuration file argument. Using this configuration file, you can specify VCS to monitor the following:

- FSMs in the design
- States and transitions between states
- Maximum number of sequences to track in each module or entity

For more information on the configuration file, refer to the VCS/
VCS MX Coverage Metrics User Guide.

- Managing Coverage databases

Typically, VCS names the coverage files in the coverage database as `test.line`, `test.cond`, and so on. Every time you run a new simulation, VCS overwrites these coverage files if they already exist. Therefore, Synopsys recommends that you specify a distinct name for the coverage files using the `-cm_name` `test_name` option during simulation. This is most useful if you follow the preferred compile once, run many times flow. For additional information, refer to the section, “[Compile Once and Run Many Times](#)” on page 10-4.

- Including Verilog libraries for coverage computation

By default, VCS will not write coverage information for your Verilog libraries or modules under `celldefine. You can use the `-cm_libs yv` compilation option to generate coverage information for your Verilog libraries, and `-cm_libs celldefine` for modules under `celldefine.

Refer to the *VCS/VCS MX Coverage Metrics User Guide* for more information.

Unified Report Generator (URG)

URG generates a combined report for all types of coverage information. These reports are organized by the design hierarchy, module lists, and coverage groups. It also provides you with the overall summary of the entire design/testbench for all tests.

By default, URG generates coverage reports in html format. However, you can choose to generate reports in text format as well. This section describes the following:

- [“Usage Model to Invoke URG”](#)
 - [“Basic URG Options”](#)
 - [“Examples”](#)
-

Usage Model to Invoke URG

To invoke URG, you need to specify one or more coverage database directories. Under default coverage options, the coverage database directory can be:

- The * .vdb directory containing group and assertion coverage data
- The * .cm directory containing code coverage data

The following command line invokes URG:

```
% urg [urg_options] -dir dir1 [dir2 ....]
```

In this example, `dir1` and `dir2` are the coverage database directories. If you specify more than one coverage database directory in the URG command line, URG merges coverage database directories, and will write a merged coverage database in the current or specified directory.

By default, the above URG command writes html report files in the `urgReport` directory in the current directory. `dashboard.html` is the main page, which gives you the overall coverage information of your entire design or the testbench. You can use any browser to view the coverage reports.

Basic URG Options

This section lists only the basic URG options (for a complete list, refer to the *URG User Guide*).

`-help | -h`

Shows command line and options supported by URG.

`-dir directory_name`

Specifies coverage data directories.

`-dbname name`

Specifies the merged database name.

`-f file_name`

Specifies multiple directories for source data in a file.

`-format text`

Generates text report files instead of HTML report files.

`-log file_name`

Sends diagnostics to *file_name* instead of to stdout/stderr.

`-metric [line+cond+fsm+tgl+assert+group]`

Limits report to specified metrics.

`-noreport`

Generates only the merged results, does not generate the reports.

`-report mydir`

Generates report in *mydir* instead of default directory.

`-map module_name`

Maps sub-hierarchy code coverage from one design to another.
This option is not available for assert or group coverage. The full hierarchy is generated in the `hierarchy.html` file.

Examples

The following examples demonstrate the use of `urg`:

Example 1: To merge coverage database directories

This example merges simv1.cm, simv2.cm and simv.vdb, and writes a report in html format in the current working directory.

```
% urg -dir simv1.cm simv2.cm simv.vdb
```

The above command line merges all the coverage data for all tests stored in directories simv1.cm, simv2.cm and simv.vdb and writes the merged report in the ./urgReport directory.

Example 2: To generate report in text format

Consider the same coverage database directories used in Example 1. Now, using the urg -format text option, you can generate reports in text format. The command line is shown below:

```
% urg -dir simv1.cm simv2.cm simv.vdb -format text
```

This command writes reports in text format in the ./urgReport directory.

13

Using OpenVera Native Testbench

OpenVera Native Testbench is a high-performance, single-kernel technology in VCS that enables:

- Native compilation of testbenches written in OpenVera and in SystemVerilog.
- Simulation of these testbenches along with the designs.

This technology provides a unified design and verification environment in VCS for significantly improving overall design and verification productivity. Native Testbench is uniquely geared towards efficiently catching hard-to-find bugs early in the design cycle, enabling not only completing functional validation of designs with the desired degree of confidence, but also achieving this goal in the shortest time possible.

Native Testbench is built around the preferred methodology of keeping the testbench and its development separate from the design. This approach facilitates development, debug, maintenance and reusability of the testbench, as well as ensuring a smooth synthesis flow for your design by keeping it clean of all testbench code. Further, you have the choice of either compiling your testbench along with your design or separate from it. The latter choice not only saves you from unnecessary recompilations of your design, it also enables you to develop and maintain multiple testbenches for your design.

This chapter describes the high-level, object-oriented verification language of OpenVera, which enables you to write your testbench in a straightforward, elegant and clear manner and at a high level essential for a better understanding of and control over the design validation process. Further, OpenVera assimilates and extends the best features found in C++ and Java along with syntax that is a natural extension of the hardware description languages. Adopting and using OpenVera, therefore, means a disciplined and systematic testbench structure that is easy to develop, debug, understand, maintain and reuse.

Thus, the high-performance of Native Testbench technology, together with the unique combination of the features and strengths of OpenVera, can yield a dramatic improvement in your productivity, especially when your designs become very large and complex.

This chapter includes the following topics:

- “Usage Model”
- “Key Features”

Usage Model

As any other VCS applications, the usage model to simulate OpenVera testbench includes the following steps:

Compilation

```
% vcs [ntb_options] [compile_options] file1.vr file2.vr  
      file3.v file4.v
```

Simulation

```
% simv [run_options]
```

Example

In this example, we have an interface file, a Verilog design `arb.v`, OpenVera testbench `arb.vr`, all instantiated in a Verilog top file, `arb.test_top.v`.

```
//Interface  
#ifndef INC_ARB_IF_VRH  
#define INC_ARB_IF_VRH  
  
interface arb {  
    input clk CLOCK;  
    output [1:0] request OUTPUT_EDGE OUTPUT_SKEW;  
    output reset OUTPUT_EDGE OUTPUT_SKEW;  
    input [1:0] grant INPUT_EDGE INPUT_SKEW;  
} // end of interface arb  
  
#endif
```

```

//Verilog module: arb.v
module arb ( clk, reset, request, grant) ;
    input [1:0] request ;
    output [1:0] grant ;
    input   reset ;
    input   clk ;

    parameter IDLE = 2, GRANT0 = 0, GRANT1 = 1;

    reg  last_winner ;
    reg winner ;
    reg [1:0] grant ;
    reg [1:0] next_grant ;

    reg [1:0] state, nxState;

    ...

endmodule

//OpenVera Testbench: arb.vr

#define OUTPUT_EDGE PHOLD
#define OUTPUT_SKEW #1
#define INPUT_SKEW #-1
#define INPUT_EDGE PSAMPLE
#include <veraDefines.vrh>

#include "arb.if.vrh"

program arb_test
{ // start of top block

    ...

} // end of program arb_test

```

Note:

You can find the complete example in \$VCS_HOME/doc/examples/nativetestbench/openvera/tutorial/arb.

Usage Model

Compilation

```
% vcs -ntb arb.v arb.vr arb.test_top.v
```

Simulation

```
% simv
```

Using Template Generator

To ease the process of writing a testbench in OpenVera, VCS provides you with a testbench template generator.

Use the following command to invoke the template generator on a Verilog design unit:

```
% ntb_template -t design_module_name [-c clock] design_file\[-vcs vcs_compile-time_options]
```

Where:

-t *design_module_name*

Specifies the top-level design module name.

design_file

Name of the design file.

-C

Specifies the clock input of the design.

-template
Can be omitted.

-program

Optional. Use it to specify program name.

-simcycle

Optional. Use this to override the default cycle value of 100.

-vcs *vcs_compile-time_options*

Optional. Use it to supply a VCS compile-time option. Multiple -vcs *vcs_compile-time_options* options can be used to specify multiple options. Use this option only for Verilog on top designs.

Example

An example SRAM model is used in this demonstration of using the template generator to develop a testbench environment.

For details on the OpenVera verification language, refer to the *OpenVera Language Reference Manual: Native Testbench*.

Design Description

The design is an SRAM whose RTL Verilog model is in the file sram.v. It has four ports:

- ce_N (chip enable)
- rdWr_N (read/write enable)
- ramAddr (address)
- ramData (data)

Example 13-1 RTL Verilog Model of SRAM in sram.v

```
module sram(ce_N, rdWr_N, ramAddr, ramData) ;  
  
    input ce_N, rdWr_N;  
    input [5:0] ramAddr;  
    inout [7:0] ramData;  
    wire [7:0] ramData;  
    reg [7:0] chip[63:0];  
  
    assign #5 ramData = (~ce_N & rdWr_N) ? chip[ramAddr] :  
        8'bzzzzzzz;  
  
    always @(ce_N or rdWr_N)  
    begin  
        if(~ce_N && ~rdWr_N)  
            #3 chip[ramAddr] = ramData;  
    end  
endmodule
```

During a read operation, when `ce_N` is driven low and `rdWr_N` is driven high, `ramData` is continuously driven from inside the SRAM with the value stored in the SRAM memory element specified by `ramAddr`. During a write operation, when both `ce_N` and `rdWr_N` are driven low, the value driven on `ramData` from outside the SRAM is stored in the SRAM memory element specified by `ramAddr`. At all other times, `ce_N` is driven high, and as a result, `ramData` gets continuously driven from inside the SRAM with the high-impedance value `Z`.

Generating the Testbench Template, the Interface, and the Top-level Verilog Module from the Design

As previously mentioned, Native Testbench provides a template generator to start the process of constructing a testbench. The template generator is invoked on `sram.v` as shown below:

```
% ntb_template -t sram sram.v
```

Where:

- The `-t` option is followed with the top-level design module name, which is `sram`, in this case.
- `sram` is the name of the module.
- `sram.v` is the name of the file containing the top-level design module.
- If the design uses a clock input, then the `-c` option is to be used and followed with the name of the clock input. Doing so provides a clock input derived from the system-clock for the interface and the design. In this example, there is no clock input required by the design.

Template generator generates the following files:

- `sram.vr.tmp`
- `sram.if.vrh`
- `sram.test_top.v`

sram.vr.tmp

This is the template for testbench development. The following is an example, based on the `sram.v` file of the output of the previous command line:

```
//sram.vr.tmp
#define OUTPUT_EDGE PHOLD
#define OUTPUT_SKEW #1
#define INPUT_SKEW #-1
#define INPUT_EDGE PSAMPLE
#include <veraDefines.vrh>

// define interfaces, and verilog_node here if necessary

#include "sram.if.vrh"

// define ports, binds here if necessary

// declare external tasks/classes/functions here if
//necessarry

// declare verilog_tasks here if necessary

// declare class typedefs here if necessary

program sram_test
{ // start of top block

    // define global variables here if necessary

    // Start of sram_test

    // Type your test program here:

    //
    // Example of drive:
    // @1 sram.ce_N = 0 ;
    //
    //
    // Example of expect:
```

```

// @1,100 sram.example_output == 0 ;
//
}

} // end of program sram_test

// define tasks/classes/functions here if necessary

```

sram.if.vrh

This is the interface file which provides the basic connectivity between your testbench signals and your design's ports and/or internal nodes. All signals going back and forth between the testbench and the design go through this interface. The following is the `sram.if.vrh` file which results from the previous command line:

```

//sram.if.vrh
#ifndef INC_SRAM_IF_VRH
#define INC_SRAM_IF_VRH
interface sram {
    output ce_N      OUTPUT_EDGE OUTPUT_SKew;
    output rdWr_N   OUTPUT_EDGE OUTPUT_SKew;
    output [5:0] ramAddr OUTPUT_EDGE OUTPUT_SKew;
    inout [7:0] ramData INPUT_EDGE INPUT_SKew
OUTPUT_EDGE OUTPUT_SKew;
} // end of interface sram

#endif

```

Notice that, for example, the direction of `ce_N` is now "output" instead of "input". The signal direction specified in the interface is from the point of view of the testbench and not the DUT.

This file must be modified to include the clock input.

sram.test_top.v

This is the top-level Verilog module that contains the testbench instance, the design instance, and the system-clock. The system clock can also provide the clock input for both the interface and the design. The following is the `sram.test_top.v` file that results from the previous command line:

```
//sram.test_top.v
module sram_test_top;
    parameter simulation_cycle = 100;

    reg SystemClock;

    wire ce_N;
    wire rdWr_N;
    wire [5:0] ramAddr;
    wire [7:0] ramData;

`ifdef SYNOPSYS_NTB
    sram_test vshell(
        .SystemClock (SystemClock),
        .\sram.ce_N (ce_N),
        .\sram.rdWr_N (rdWr_N),
        .\sram.ramAddr (ramAddr),
        .\sram.ramData (ramData)
    );
`else

    vera_shell vshell(
        .SystemClock (SystemClock),
        .sram_ce_N (ce_N),
        .sram_rdWr_N (rdWr_N),
        .sram_ramAddr (ramAddr),
        .sram_ramData (ramData)
    );
`endif

`ifdef emu
/* DUT is in emulator, so not instantiated here */
`else
    sram dut(

```

```

        .ce_N      (ce_N),
        .rdWr_N   (rdWr_N),
        .ramAddr  (ramAddr),
        .ramData   (ramData)
    );
`endif

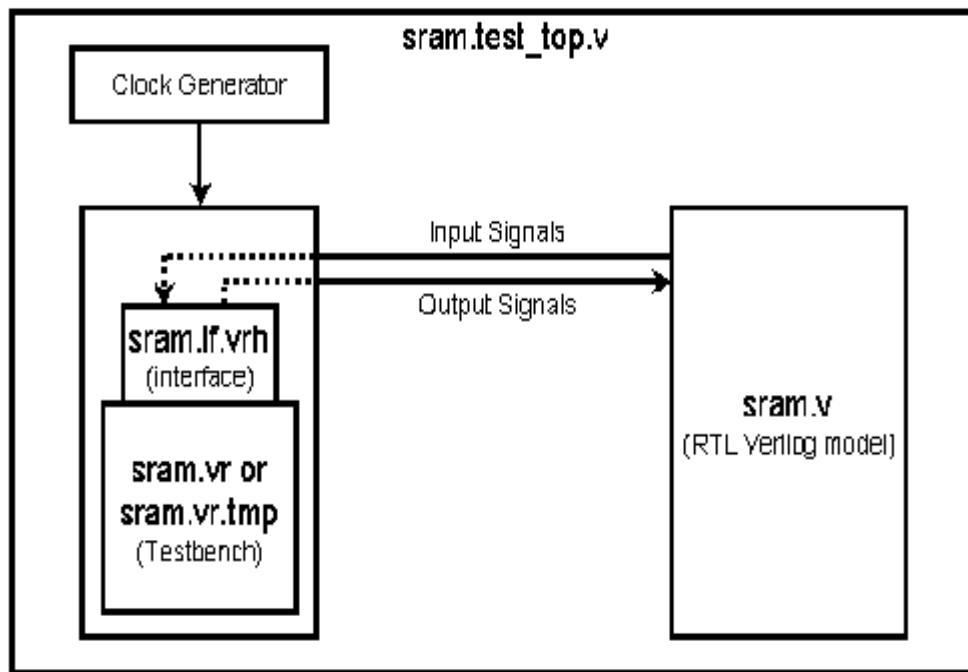
initial begin
    SystemClock = 0;
    forever begin
        #(simulation_cycle/2)
        SystemClock = ~SystemClock;
    end
end

endmodule

```

[Figure 13-1](#) shows how the three generated files and the design connect and fit in with each other in the final configuration.

Figure 13-1 Testbench and Design Configuration



Testbench Development and Description

Your generated testbench template, `sram.vr.tmp`, contains a list of macro definitions for the interface, include statements for the interface file and the library containing predefined tasks and functions, comments indicating where to define or declare the various parts of the testbench, and the skeleton program shell that will contain the main testbench constructs. Starting with this template, you can develop a testbench for the SRAM and rename it `sram.vr`. An example testbench is shown in [Example 13-2](#).

Example 13-2 Example testbench for SRAM, sram.vr

```
// macro definitions for Interface signal types and skews
#define OUTPUT_EDGE PHOLD      // for specifying posedge-drive type
#define OUTPUT_SKEW #1         // for specifying drive skew
#define INPUT_SKEW #-1        // for specifying sample skew
#define INPUT_EDGE PSAMPLE    // for specifying posedge-sample type

#include <veraDefines.vrh>      // include the library of predefined
                                // functions and tasks
#include "sram.if.vrh"          // include the Interface file

program sram_test {           // start of program sram_test

    reg [5:0] address = 6'b00_0001; // declare, initialize address (for
                                    // driving ramAddr during Write and
                                    // Read)
    reg [7:0] rand_bits;          // declare rand_bits (for driving
                                    // ramData during Write)
    reg [7:0] data_result;        // declare data_result (for receiving
                                    // ramData during Read)

    @(posedge sram.clk);
    rand_bits = random();        // move to the first posedge of clock
                                // initialize rand_bits with a random
                                // value using the random() function

    @1 sram.ramAddr = address;   // move to the next posedge of clock,
                                // drive ramAddr with the value of
                                // address
    sram.ce_N = 1'b1;            // disable SRAM by driving ce_N high
    sram.ramData = rand_bits;    // drive ramData with rand_bits and
                                // keep it ready for a Write
    sram.rdWr_N = 1'b0;          // drive rdWr_N low and keep it ready
                                // for a Write
```

```

@1 sram.ce_N = 1'b0;           // move to the next posedge of clock,
                                // and enable a SRAM Write by driving
                                // ce_N low
printf("Cycle: %d Time: %d \n", get_cycle(), get_time(0));
printf("The SRAM is being written at ramAddr: %b Data written: %b \n", address,
sram.ramData);

@1 sram.ce_N = 1'b1;           // move to the next posedge of clock,
                                // disable SRAM by driving ce_N high
sram.rdWr_N = 1'b1;           // drive rdWr_N high and keep it ready
                                // for a Read
sram.ramData = 8'bzzzz_zzzz;  // drive a high-impedance value on
                                // ramData

@1 sram.ce_N = 1'b0;           // move to the next posedge of clock,
                                // enable a SRAM Read by driving ce_N
                                // low

@1 sram.ce_N = 1'b1;           // move to the next posedge of clock,
                                // disable SRAM by driving ce_N high
data_result = sram.ramData;    // sample ramData and receive the data
                                // from SRAM in data_result
printf("Cycle: %d Time: %d\n", get_cycle(), get_time(0));
printf("The SRAM is being read    at ramAddr: %b Data read    : %b \n", address,
data_result);

} // end of program sram_test

```

The main body of the testbench is the program, which is named `sram_test`. The program contains three data declarations of type `reg` in the beginning. It then moves execution through a Write operation first and then a Read operation. The memory element of the SRAM written to and read from is `6'b 00_0001`. The correct functioning of the SRAM implies data that is stored in a memory element during a Write operation must be the same as that which is received from the memory element during a Read operation later. The example testbench only demonstrates how any memory element can be functionally validated. For complete functional validation of the SRAM, the testbench would need further development to cover all memory elements from `6'b00_0000` to `6b'11_1111`.

Interface Description

The generated `if.vrh` file has to be modified to include the clock input. The modified interface is shown in [Example 13-3](#).

Interface for SRAM, `sram.if.vrh`

Example 13-3

```
#ifndef INC_SRAM_IF_VRH
#define INC_SRAM_IF_VRH

interface sram {
    input          clk      CLOCK; // add clock
    output         ce_N    OUTPUT_EDGE OUTPUT_SKew;
    output         rdWr_N  OUTPUT_EDGE OUTPUT_SKew;
    output [5:0]   ramAddr OUTPUT_EDGE OUTPUT_SKew;
    inout [7:0]    ramData INPUT_EDGE  OUTPUT_EDGE OUTPUT_SKew;
} // end of interface sram
#endif
```

The interface consists of signals that are either driven as outputs into the design or sampled as inputs from the design. The clock input, `clk`, is derived from the system clock in the top-level Verilog module.

Top-level Verilog Module Description

The generated top-level module has been modified to include the clock input for the interface and eliminate code that was not relevant. The clock input is derived from the system clock. [Example 13-4](#) shows the modified top-level Verilog module for the SRAM.

Example 13-4 Top-level Verilog Module, sram.test_top.v

```
module sram_test_top;
    parameter simulation_cycle = 100;
    reg           SystemClock;
    wire          ce_N;
    wire          rdWr_N;
    wire [5:0]    ramAddr;
    wire [7:0]    ramData;
    wire          clk = SystemClock; /* Add this line. Interface
                                      clock input derived from the system clock*/
`ifdef SYNOPSYS_NTB
    sram_test vshell(
        .SystemClock (SystemClock),
        .\sram.clk(clk),
        .\sram.ce_N (ce_N),
        .\sram.rdWr_N      (rdWr_N),
        .\sram.ramAddr     (ramAddr),
        .\sram.ramData     (ramData)
    );
`else
    vera_shell vshell(
        .SystemClock (SystemClock),
        .sram_ce_N   (ce_N),
        .sram_rdWr_N (rdWr_N),
        .sram_ramAddr (ramAddr),
        .sram_ramData (ramData)
    );
`endif

// design instance
sram dut(
    .ce_N       (ce_N),
    .rdWr_N    (rdWr_N),
    .ramAddr   (ramAddr),
    .ramData   (ramData)
);

// system-clock generator
initial begin
```

```

SystemClock = 0;
forever begin
    #(simulation_cycle/2)
        SystemClock = ~SystemClock;
    end
end

endmodule

```

The top-level Verilog module contains the following:

- The system clock, `SystemClock`. The system clock is contained in the port list of the testbench instance.
- The declaration of the interface clock input, `clk`, and its derivation from the system clock.
- The testbench instance, `vshell`. The module name for the instance must be the name of the testbench program, `sram_test`. The instance name can be something you choose. The ports of the testbench instance, other than the system clock, refer to the interface signals. The period in the port names separates the interface name from the signal name. A backslash is appended to the period in each port name because periods are not normally allowed in port names.
- The design instance, `dut`.

Compiling Testbench With the Design And Running

The VCS command line for compiling both your example testbench and design is the following:

Compilation

```
% vcs -ntb sram.v sram.test_top.v sram.vr
```

Simulation

```
% simv
```

You will find the simulation output to be the following:

```
Cycle: 3 Time: 250
The SRAM is being written at ramAddr: 000001 with ramData:
10101100
Cycle: 6 Time: 550
The SRAM is being read at ramAddr: 000001 its ramData is:
10101100
$finish at simulation time      550
V C S   S i m u l a t i o n   R e p o r t
```

Key Features

VCS supports the following features for OpenVera testbench:

- “Multiple Program Support”
- “Separate Compilation of Testbench Files”
- “Class Dependency Source File Reordering”
- “Using Encrypted Files”
- “Functional Coverage”
- “Using Reference Verification Methodology”
- “Performance Profiler”
- “Memory Profiler”

Multiple Program Support

Multiple program support enables multiple testbenches to run in parallel. This is useful when testbenches model stand-alone components (for example, Verification IP (VIP) or work from a previous project). Because components are independent, direct communication between them except through signals is undesirable. For example, UART and CPU models would communicate only through their respective interfaces, and not via the testbench. Thus, multiple program support allows the use of stand-alone components without requiring knowledge of the code for each component, or requiring modifications to your own testbench.

Configuration File Model

The configuration file that you create, specifies file dependencies for OpenVera programs.

Specify the configuration file as an argument to `-ntb_opts` as shown in the following usage model:

```
% vcs -ntb -ntb -ntb_opts  
config=configfileVerilog_and_OV_files
```

Configuration File

The configuration file contains the program construct.

The program keyword is followed by the OpenVera program file (.vr file) containing the testbench program and all the OpenVera program files needed for this program. For example:

```
//configuration file
program
    main1.vr
    main1_dep1.vr
    main1_dep2.vr
    ...
    main1_depN.vr
    [NTB_options ]

program
    main2.vr
    main2_dep1.vr
    main2_dep2.vr
    ...
    main2_depN.vr
    [NTB_options ]

program
    mainN.vr
    mainN_dep1.vr
    mainN_dep2.vr
    ...
    mainN_depN.vr
    [NTB_options ]
```

In this example, main1.vr, main2.vr and mainN files each contain a program. The other files contain items such as definitions of functions, classes, tasks and so on needed by the program files. For example, the main1_dep1.vr, main1_dep2.vr main1_depN.vr files contain definitions relevant to main1.vr. Files main2_dep1.vr, main2_dep2.vr ... main2_depN.vr contain definitions relevant to main2.vr, and so forth.

Usage Model for Multiple Programs

You can specify programs and related support files with multiple programs in two different ways:

1. Specifying all OpenVera programs in the configuration file
2. Specifying one OpenVera program on the command line, and the rest in the configuration file

Note:

- Specifying multiple OpenVera files containing the program construct at the VCS command prompt is an error.
- If you specify one program at the VCS command line and if any support files are missing from the command line, VCS issues an error.

Specifying all OpenVera programs in the configuration file

When there are two or more program files listed in the configuration file, the VCS command line is:

```
% vcs -ntb -ntb_opts config=configfile
```

The configuration file, could be:

```
program main1.vr -ntb_define ONE
program main2.vr -ntb_incdir /usr/vera/include
```

Specifying one OpenVera program on the command line, and the rest in the configuration file

You can specify one program in the configuration file and the other program file at the command prompt.

```
% vcs -ntb -ntb_opts config=configfile main2.vr
```

The configuration file used in this example is:

```
program main1.vr
```

In the previous example, `main1.vr` is specified in the configuration file and `main2.vr` is specified on the command line along with the files needed by `main2.vr`.

NTB Options and the Configuration File

The configuration file supports different OpenVera programs with different NTB options such as '`include`', '`define`', or '`timescale`'. For example, if there are three OpenVera programs `p1.vr`, `p2.vr` and `p3.vr`, and `p1.vr` requires the `-ntb_define VERA1` runtime option, and `p2.vr` should run with `-ntb_incdir /usr/vera/include` option, specify these options in the configuration file:

```
program p1.vr -ntb_define VERA1
program p2.vr -ntb_incdir /usr/vera/include
```

and specify the command line as follows.

```
% vcs -ntb -ntb_opts config=configfile p3.vr
```

Any NTB options mentioned at the command prompt, in addition to the configuration file, are applicable to all OpenVera programs.

In the configuration file, you may specify the NTB options in one line separated by spaces, or on multiple lines.

```
program file1.vr -ntb_opts no_file_by_file_pp
```

Some NTB options specific for OpenVera code compilation, such as `-ntb_cmp` and `-ntb_vl`, affect the VCS flow after the options are applied. If these options are specified in the configuration file, they are ignored.

The following options are allowed for multiple program use.

- `-ntb_define macro`
- `-ntb_incdir directory`
- `-ntb_opts no_file_by_file_pp`
- `-ntb_opts tb_timescale=value`
- `-ntb_opts dep_check`
- `-ntb_opts print_deps`
- `-ntb_opts use_sigprop`
- `-ntb_opts vera_portname`

See and “Compile-time Options” for the description of the these options.

Separate Compilation of Testbench Files

This section describes how to compile your testbench separately from your design and then load it on simv (compiled design executable) at runtime. Separate compilation of testbench files allows you to:

- Keep one or many testbenches compiled and ready and then choose which testbench to load when running a simulation.

- Save time by recompiling only the testbench after making changes to it and then running simv with the recompiled testbench.
- Save time in cases where changes to the design do not require changes to the testbench by recompiling only the design after making changes to it and then running simv with the previously compiled testbench.

Separate compilation of the testbench generates two files:

- The compiled testbench in a shared object file, `libtb.so`. This shared object file is the one to be loaded on simv at runtime.
- A Verilog shell file (`.vshell`) that contains the testbench shell module. Since the testbench instance in the top-level Verilog module now refers to this shell module, the shell file has to be compiled along with the design and the top-level Verilog module. The loaded shared object testbench file is automatically invoked by the shell module during simulation.

The following steps demonstrate a typical flow involving separate compilation of the testbench:

1. Compile the testbench in VCS to generate the shared object (`libtb.so`) file containing the compiled testbench and the Verilog testbench shell file.
2. Compile the HDL along with the top-level Verilog module and the testbench shell (`.vshell`) file to generate the executable simv.
3. Load the testbench on simv at runtime.

Important:

The following `ntb_opts` options must be used for both steps of the compilation (the testbench compilation and the design compilation):

```
-ntb_opts use_sigprop  
-ntb_opts dw_vip  
-ntb_opts aop
```

Usage Model

Testbench Compilation

```
% vcs -ntb_cmp [other_ntb_options] file1.vr file2.vr
```

Compilation

```
% vcs -ntb_vl [ntb_options] [compile_options]  
file1.vr file2.vr file3.v file4.v
```

Simulation

```
% simv +ntb_load=PATH/libtb.so [run_options]
```

PATH is the directory where the `libtb.so` and `.vshell` files are created. You can specify *PATH* by using the `-ntb_spath` option while compiling the testbench.

Example

Design files: `top.v` `mid.v`, `bot.v`

Testbench file: tb.vr

```
% vcs -ntbmx_cmp -timescale=1ns/1ps tb.vr  
  
% vcs -ntb_vl -timescale=1ns/1ps top.v mid.v bot.v  
  
% simv +ntb_load=./libtb.so
```

Class Dependency Source File Reordering

In order to ease transitioning of legacy code from Vera's make-based single-file compilation scheme to VCS-NTB, where all source files have to be specified on the command line, VCS provides a way of instructing the compiler to reorder Vera files in such a way that class declarations are in topological order (that is, base classes precede derived classes).

In Vera, where files are compiled one-by-one, and extensive use of header files is a must, the structure of file inclusions makes it very likely that the combined source text has class declarations in topological order.

If specifying a command line like the following leads to problems (error messages related to classes), adding the analysis option `-ntb_opts dep_check` to the command line directs the compiler to activate analysis of Vera files and process them in topological order with regard to class derivation relationships.

```
% vcs -ntb *.vr
```

By default, files are processed in the order specified (or wildcard-expanded by the shell). This is a global option, and affects all Vera input files, including those preceding it, and those named in `-f file.list`.

When using the option `-ntb_opts print_deps` in addition to `-ntb_opts dep_check` with vcs, the reordered list of source files is printed on standard output. This could be used, for example, to establish a baseline for further testbench development.

For example, assume the following files and declarations:

```
b.vr: class Base {integer i;}  
d.vr: class Derived extends Base {integer j;}  
p.vr: program test {Derived d = new;}
```

File `d.vr` depends on file `b.vr`, since it contains a class derived from a class in `b.vr`, whereas `p.vr` depends on neither, despite containing a reference to a class declared in the former. The `p.vr` file does not participate in inheritance relationships. The effect of dependency ordering is to properly order the files `b.vr` and `d.vr`, while leaving files without class inheritance relationships alone.

The following command lines result in reordered sequences.

```
% vcs -ntb -ntb_opts dep_check d.vr b.vr p.vr  
% vcs -ntb -ntb_opts dep_check p.vr d.vr b.vr
```

The first command line yields the order `b.vr d.vr p.vr`, while the second line yields, `p.vr b.vr d.vr`.

Circular Dependencies

With some programming styles, source files can appear to have circular inheritance dependencies in spite of correct inheritance trees being cycle-free. This can happen, for example, in the following scenario:

```
a.vr: class Base_A {...}  
      class Derived_B extends Base_B {...}  
b.vr: class Base_B {...}  
      class Derived_A extends Base_A {...}
```

In this example, classes are derived from base classes that are in the other file, respectively, or more generally, when the inheritance relationships project onto a loop among the files. This is, however, an abnormality that should not occur in good programming styles. VCS will detect and report the loop, and will use a heuristic to break it. This may not lead to successful compilation, in which case you can use the `-ntb_opts print_deps` option to generate a starting point for manual resolution; however, if possible, the code should be rewritten.

Dependency-based Ordering in Encrypted Files

As encrypted files are intended to be mostly self-contained library modules that the testbench builds upon, they are excluded from reordering regardless of dependencies (these files should not exist in unencrypted code). VCS splits Vera input files into those that are encrypted or declared as such by having the `.vrp` or `.vrhp` file extension or as specified using the `-ntb_vipext` option, and others. Only the latter unencrypted files are subject to dependency-based reordering, and encrypted files are prefixed to them.

Note:

The `-ntb_opts dep_check` compile-time option specifically resolves dependencies involving classes and enums. That is, we only consider definitions and declarations of classes and enums. Other constructs such as ports, interfaces, tasks and functions are not currently supported for dependency check.

Using Encrypted Files

VCS NTB allows distributors of Verification IP (Intellectual Property) to make testbench modules available in encrypted form. This enables the IP vendors to protect their source code from reverse-engineering. Encrypted testbench IP is regular OpenVera code, and is not subject to special processing other than to protect the source code from inspection in the debugger, through the PLI, or otherwise.

Encrypted code files provided on the command line are detected by VCS, and are combined into one preprocessing unit that is preprocessed separately from unencrypted files, and is for itself, always preprocessed in `-ntb_opts no_file_by_file_pp` mode. The preprocessed result of encrypted code is prefixed to preprocessed unencrypted code.

VCS only detects encrypted files on the command line (including `-f` option files), and does not descend into include hierarchies. While the generally recommended usage methodology is to separate encrypted from unencrypted code, and not include encrypted files in unencrypted files, encrypted files can be included in unencrypted files if the latter are marked as encrypted-mode by naming them with extensions `.vrp`, `.vrhp`, or additional extensions specified using the `-ntb_vipext` option. This implies that the extensions are

considered OpenVera extensions similar to using `-ntb_fileext` for unencrypted files. This causes those files and everything they include to be preprocessed in encrypted mode.

Functional Coverage

The VCS implementation of OpenVera supports the `covergroup` construct. Covergroups are specified by the user. They allow the system to monitor values and transitions for variables and signals. They also enable cross coverage between variables and signals.

If you have covergroups in your design, VCS collects the coverage data during simulation and generates a database, `simv.vdb`. Once you have `simv.vdb`, you can use Unified Report Generator to generate text or HTML reports (see “[Unified Report Generator \(URG\)](#)” on page 12-13).

Unified Coverage Directory and Database Control

A coverage directory named `simv.vdb` contains all the testbench functional coverage data. This is different from previous versions of VCS, where the coverage database files were stored by default in the current working directory or the path specified by `coverage_database_filename`. For your reference, VCS associates a logical test name with the coverage data that is generated by a simulation. VCS assigns a default test name; you can override this name by using the `coverage_set_test_database_name` task.

```
task coverage_set_test_database_name  
    ("test_name" [, "dir_name"] ) ;
```

VCS avoids overwriting existing database file names by generating unique non-clashing test names for consecutive tests.

For example, if the coverage data is to be saved to a test name called `pci_test`, and a database with that test name already exists in the `simv.vdb` coverage directory, then VCS automatically generates the new name `pci_test_gen1` for the next simulation run. The following table explains the unique name generation scheme details.

Table 13-1 Unique Name Generation Scheme

Test Name	Database for the...
<code>pci_test</code>	1st testbench run
<code>pci_test_gen_1</code>	2nd testbench run
<code>pci_test_gen_2</code>	3rd testbench run
<code>pci_test_gen_n</code>	n th testbench run

You can disable this method of ensuring database backup and force VCS to always overwrite an existing coverage database. To do this, use the following system task:

```
task coverage_backup_database_test (flag);
```

The value of flag can be:

- OFF for disabling database backup.
- ON for enabling database backup.

In order to not save the coverage data to a database file (for example, if there is a verification error), use the following system task:

```
task coverage_save_database (flag);
```

The value of flag can be:

- OFF for disabling database backup.
- ON for enabling database backup.

Loading Coverage Data

Both cumulative coverage data and instance-specific coverage data can be loaded. The loading of coverage data from a previous VCS run implies that the bin hits from the previous VCS run to this run are to be added.

Loading Cumulative Coverage Data

The cumulative coverage data can be loaded either for all coverage groups, or for a specific coverage group. To load the cumulative coverage data for all coverage groups, use the following syntax:

```
coverage_load_cumulative_data("test_name" [, "dir_name"]);
```

In this task, "dir_name" is optional. If you do not specify a "dir_name", by default, simv.vdb is taken as the directory containing the database.

The above task directs VCS to find the cumulative coverage data for all coverage groups found in the specified database file and to load this data if a coverage group with the appropriate name and definition exists in this VCS run.

To load the cumulative coverage data for just a single coverage group, use the following syntax:

```
coverage_load_cumulative_cg_data("test_name",
                                  "covergroup_name" [, "dir_name"]);
```

In this task, "dir_name" is optional. If you do not specify a "dir_name", by default, simv.vdb is taken as the directory containing the database.

In the following example, a Vera class, MyClass, exists with an embedded coverage object covType. VCS finds the cumulative coverage data for the coverage group MyClass:covType in the database file Run1 and loads it into the covType embedded coverage group in MyClass.

Example 13-5

```
MyClass {
    integer m_e;
    coverage_group covType{
        sample_event = wait_var(m_e);
        sample m_e;
    }
}
...
...
coverage_load_cumulative_cg_data("Run1",
                                  "MyClass::covType");
```

Loading Instance Coverage Data

The coverage data can be loaded for a specific coverage instance. To load the coverage data for a stand-alone coverage instance, use the following syntax:

```
coverage_instance.load("test_name" [, "dir_name"]);
```

In this task, dir_name is optional. If you do not specify a dir_name, by default, simv.vdb is taken as the directory containing the database.

To load the coverage data for an embedded coverage instance, use the following syntax:

```
class_object.cov_group_name.load("test_name" [, "dir_name"]);
```

In this task, `dir_name` is optional. If you do not specify a `dir_name`, by default, `simv.vdb` is taken as the directory containing the database.

The commands shown above direct VCS to find the coverage data for the specified instance name in the database, and load it into the coverage instance.

In [Example 13-5](#), there is a Vera class `MyClass` with an embedded coverage object `covType`. Two objects `obj1` and `obj2` are instantiated, each with the embedded coverage group `covType`. VCS will find the coverage information for the coverage instance `obj1:covType` from the database file `Run1`, and load this coverage data into the newly-instantiated `obj1` object. Note that the object, `obj2`, will not be affected as part of this load operation.

Example 13-6

```
MyClass {
    integer m_e;
    coverage_group covType {
        sample_event = wait_var(m_e);
        sample m_e;
    }
}
...
...
MyClass obj1 = new;
obj1.load("Run1");
MyClass obj2 = new;
```

Note:

The compile time or runtime options `-cm_dir` and `-cm_name` will overwrite the calls to `coverage_set_test_database_name` and load the coverage data tasks.

Using Reference Verification Methodology

VCS supports the use of Reference Verification Methodology (RVM) for implementing testbenches as part of a scalable verification architecture.

The usage model for using RVM with VCS is:

Compilation

```
% vcs -ntb -ntb_opts rvm [ntb_options] [compile_options]  
file1.vr file2.vr file3.v file4.v
```

Simulation

```
% simv [run_options]
```

For details on the use of RVM, see the *Reference Verification Methodology User Guide*. Though the manual descriptions refer to Vera, NTB uses a subset of the OpenVera language and all language specific descriptions apply to NTB.

Differences between the usage of NTB and Vera are:

- NTB does not require header files (`.vrh`) as described in the *Reference Verification Methodology User Guide* chapter “Coding and Compilation.”
- NTB parses all testbench files in a single compilation.

- The VCS command-line option `-ntb_opts rvm` must be used with NTB.

Limitations

- The handshake configuration of notifier is not supported (since there is no handshake for triggers/syncs in NTB).
- RVM enhancements for assertion support in Vera 6.2.10 and later are not supported for NTB.
- If there are multiple consumers and producers, there is no guarantee of fairness in reads from channels, etc.

Performance Profiler

The NTB Performance Profiler aids in writing more efficient OpenVera code. When performance profiling is turned on, NTB tracks:

- The time spent in each function, task and program
- The time spent in the HDL simulator and in testbench internal
- Predefined methods
- OpenVera fork join block
- Predefined procedures
- Testbench garbage collection

Enabling the NTB Profiler

The VCS profiler has been enhanced to support NTB. The `+prof` option enables the profiling of OpenVera NTB constructs and is used in the `vcs` command line. For example:

```
% vcs +prof [ntb_options] [compile_options] Verilog_files
```

The NTB profile report is dumped in the `vcs.prof` log file.

Performance Profiler Example

The NTB performance profiler is illustrated here by means of a simple example. The program, `MyTest`, calls an OpenVera task entitled `MyPack`, and a DPI task entitled `DPI_call`, together in a loop 20 times. The profiler reports the relative portion of the runtime that each consumes.

```
//OpenVera: dpi.vr
#include <veraDefines.vrh>

// declare DPI tasks
import "DPI" function void DPI_call(integer a, integer b);

class A {
    packed rand integer i;
}

task MyPack(integer k){
    bit[31:0] arr[];
    integer result, index, left,right;
    A a = new;
    a.i = k;
    index = 0;
    left = 0;

    right = 0;
    result = 0;
    result = a.pack(arr,index,left,right,0);
}

program MyTest
{
    integer k = 0, j = 0;
    repeat (20)
    {
        fork
        {
```

```

        for (j = 0; j < 200; j++)
    {
        k = k + 1;
        MyPack(k);
    }
}
{
    for(j = 1; j < 100000; j++)
        k++;
    DPI_call(1, 2);
}
join all
}

//C-file: dpi.c
#include <svdpi.h>

static int tmp;
static void do_activity(int j)
{
    int i;

    for( i = 0; i < 200000; i++ )
    {
        tmp ^= j + i;
    }
}
void DPI_call(int a, int b)
{
    int i;

    for( i = 0; i < 1000; i++ )
    {
        i +=b;
        i *=a;
        do_activity( i );
    }
}

```

The usage model to simulate this design is shown below:

Compilation

```
% vcs +prof -ntb sram.v sram.test_top.v sram.vr
```

Simulation

```
% simv
```

Example 13-7 Log File (vcs.prof)

```
//      Synopsys VCS Y-2006.06-SP1-5
//      Simulation profile: vcs.prof
//      Simulation Time:      18.640 seconds

=====
          TOP LEVEL VIEW
=====

      TYPE      %Totaltime
-----
          DPI      97.98
          PLI      0.00
          VCD      0.00
          KERNEL   0.00
          MODULES  0.00
          PROGRAMS 1.90
          PROGRAM GC 0.11
          ASSERTION KERNEL 0.00
-----

Reporting limit is 0.50%
=====

          MODULE VIEW
=====

  (index) Module      %%Totaltime    No of Instances     Definition
-----
-----
```



```
=====
          PROGRAM VIEW
=====
  Program(index)      %Totaltime    No of Instances     Definition
-----
```

```
-----  
MyTest_p (1) 1.90 2 dpi.vr:26.  
-----
```

```
=====  
 INSTANCE VIEW  
=====  
 Instance %Totaltime  
-----  
 MyTest 1.90  
-----
```

```
=====  
 PROGRAM TO CONSTRUCT MAPPING  
=====
```

```
1. MyTest_p  
-----  
Construct Construct type %Totaltime %Programtime LineNo  
-----  
Fork Program Thread 0.90 47.06 dpi.vr : 39-43.  
A::pack Object Pack 0.84 44.12 dpi.vr : 23.  
MyPack Program Task 0.17 8.82 dpi.vr : 11-23.
```

```
=====  
 TOP-LEVEL CONSTRUCT VIEW  
-----  
 Construct %Totaltime  
-----  
 Program Thread 0.90  
 Object Pack 0.84  
 Program Task 0.17
```

```
=====  
 CONSTRUCT VIEW ACROSS DESIGN  
=====
```

1. Program Thread

Program	%TotalTime
MyTest_p	0.90

2. Object Pack

Program	%TotalTime
MyTest_p	0.84

3. Program Task

Program	%TotalTime
---------	------------

```
=====// Simulation memory: 6316085 bytes=====
```

```
=====TOP LEVEL VIEW=====
```

TYPE	Memory	%Totalmemory
DPI	2068	0.03
PLI	0	0.00
VCD	0	0.00
KERNEL	1638725	25.95
MODULES	0	0.00
PROGRAMS	9349992	148.03

```
=====
```

```

PROGRAM VIEW
=====
Program(index)          Memory   %Totalmemory No of Instances  Definition
-----
MyTest_p                (1)    9349992        148.03           2             dpi.vr:26.
-----
*****End of vcs.prof*****

```

The `vcs.prof` log file shown in Example 13-7 provides the following information:

- The DPI function in `dpi.c` consumed 97.98% of the total time.

Fork	Program Thread	0.90	47.06	dpi.vr : 39-43.
A::pack	Object Pack	0.84	44.12	dpi.vr : 23
MyPack	Program Task	0.17	8.82	dpi.vr : 11-23

- The fork-join block defined in `test.vr:39-43` consumed 0.90% of the total time.
- The predefined class method, `pack()`, invoked at `test.vr:23` consumed 0.84% of the total time.
- The task `MyPack()` defined at `test.vr:11:23` consumed 0.17% of the total time.

The time reported for construct is the exclusive time consumed by the construct itself. Time spent in dependants is not reported.

Memory Profiler

The VCS memory profiler is a Limited Customer Availability (LCA) feature. To enable this LCA feature, you must use the `-lca` compile-time option.

VCS has been enhanced to support profiling of memory consumed by the following dynamic data types:

- associative Array
- dynamic Array
- smart Queue
- string
- event
- class

This tool is available to both NTB SV and OV users.

Usage Model

The `$vcsmemprof()` task can be called from the UCLI interface. The syntax for `$vcsmemprof()` is as follows:

```
$vcsmemprof ("filename", "w|a+");
```

Where:

`filename`

Name of the file where the memory profiler dumps the report.

`w | a+`

`w` and `a+` designate the mode in which the file is opened. Specify `w` for writing and `a+` for appending to an existing file.

UCLI Interface

Compile-time

The dynamic memory profiler is enabled only if you specify `+dmprof` on the VCS compile-time command line:

```
% vcs +dmprof -ntb -debug|-debug_all OV_and_Verilog_files
```

Runtime

At runtime, invoke `$vcsmemprof()` from the UCLI command line prompt as follows:

```
simv -ucli //Invokes the ucli prompt  
ucli>call {$vcsmemprof("memprof.txt", "w|a+")}
```

Each invocation of `$vcsmemprof()` appends the profiler data to the user-specified file. The time at which the call is made is also reported. This enables you to narrow down the search for any memory issues.

The memory profiler reports the memory consumed by all the active instances of the different dynamic data types. As noted above, the memory profiler report is dumped in the *filename* specified in the `$vcsmemprof()` call.

VCS Dynamic Memory Profile Report

The memory profiler reports only memory actively held at the current simulation time instant by the dynamic data types.

Consider the following OpenVera program:

```
task t1() {
    integer arr1[*];
    arr1 = new[500];
    delay(5);
}

task t2() {
    integer arr2[*];
    arr2 = new[500];
    delay(10);
}

program main {
    fork
    {
        t1();
    }
    {
        t2();
    }
    join all
}
```

In this program, if `$vcsmemprof()` is called between 0 and 4 ns, then both `arr1` and `arr2` are active. If the call is made between 5 and 10 ns, then only `arr2` is active, and after 10 ns, neither is active.

The profile report includes the following sections.

1. Top-level view

Reports the total dynamic memory consumed in all the programs (SV/OV) and that consumed in all the modules (SV) in the system.

2. Module View

Reports the dynamic memory consumed in each SV module in the system.

3. Program View

Reports the dynamic memory consumed in each SV/OV program in the system.

4. Program-To-Construct View and Module-To-Construct View

- Task-Function-Thread section

Reports the total dynamic memory in each active task, function and thread in the module/program.

- Class Section

Reports the total dynamic memory consumed in each class in the module/program.

- Dynamic data Section

Reports the total memory consumed in each of the dynamic testbench data types - associative arrays, dynamic arrays, queues, events, strings, in the module/program.

Example 13-8

```
class FirstClass{
    integer b;
}

class SecondClass {
    integer c;
    integer d[10];
}

task FirstTask() {
    FirstClass a ;
    a = new;
    delay(100);
}

task SecondTask() {
    FirstClass a ;
    SecondClass b ;
    a = new;
    b = new;
    delay(100);
}

program test {
    integer i;
    integer sqProgram[$];
    integer sqFork[$];
    nonBlockTest();

    fork
    {
        FirstTask();
    }
    {
        delay(10);
        FirstTask();
    }
}
```

```

{
    delay(10);
    SecondTask();
}
{
    delay(20);
    sqFork.push_front(1);
    delay(120);
}
join all
sqProgram.push_front(1);
}

```

Compilation

```
% vcs test.vr +dmprof -debug_all
```

Simulation

```

% simv -ucli
ucli> next
ucli> next
ucli> call {$vcsmemprof("memprof.txt", "w") }

```

VCS Memory Profiler Output

```
=====
$vcsmemprof called at simulation time = 20
=====

=====
TOP LEVEL VIEW
=====
TYPE           MEMORY          %TOTALMEMORY
-----
MODULES         0              0.00
PROGRAMS       512             100.00
-----

=====
PROGRAM VIEW
=====
Program(index)   Memory    %TotalMemory   No of Instances  Definition
-----
test_1(1)        512        100.00        1               test.vr:30.
-----

=====
PROGRAM TO CONSTRUCT MAPPING
=====

1. test_1
-----
Task-Function-Thread
-----
Name      Type           Memory %Total #No Of      #Active  Definition
                           Memory Instances Instances
-----
FirstTask Program Task    48     9.38   1          1       test.vr:10-14
      Fork Program Thread 0     0.00   1          1       test.vr:38-39
      Fork Program Thread 0     0.00   1          1       test.vr:38-39
      test Program block 0     0.00   1          1       test.vr:30-561
-----

-----
Class Data
-----
Name      Memory %Total #objects      #objects      Allocated at
                           Memory allocated active
-----
FirstClass 48     9.38    2            31           test.vr:13
                                         test.vr:21
-----
```

Dynamic Data				
Type	Memory	%TotalMemory	#Alive Instances	
Events	336	12.32	6	
Queues	128	25.00	2	

14

Using SystemVerilog

VCS supports the SystemVerilog language as defined in the IEEE 1800-2005 standard. For information on SystemVerilog constructs, see the *SystemVerilog Language Reference Manual*.

This chapter describes the following:

- “Usage Model”
- “Using VMM with SV”
- “Debugging SystemVerilog Designs”
- “Functional Coverage”
- “Debugging SystemVerilog Designs”
- “Functional Coverage”
- “Memory Profiler”

- “[Extensions to SystemVerilog](#)”

For SystemVerilog assertions, see [Chapter 20, "Using SystemVerilog Assertions"](#).

Usage Model

The usage model to compile and simulate your design with SystemVerilog files is as follows:

Compilation

```
% vcs -sverilog [compile_options] Verilog_files
```

Simulation

```
% simv [simv_options]
```

To analyze SV files, use the option `-sverilog` with `vcs` as shown in the above usage model.

Using VMM with SV

The usage model to use VMM with SV is as follows:

Compilation

```
% vcs -sverilog -ntb_opts rvm [compile_options]  
Verilog_files
```

Simulation

```
% simv [simv_options]
```

To analyze SV files using VMM, use the option
-sverilog and -ntb_opts rvm with vcs as shown in the above usage model.

For more information on VMM, refer to the *Verification Methodology Manual for SystemVerilog*.

Debugging SystemVerilog Designs

VCS provides UCLI commands to perform the following tasks to debug a design:

Task	Related UCLI commands are...
Line stepping	step next run
Thread debugging	step thread
Setting breakpoints	stop run
Mailbox related information	show
Semaphore related information	show

For detailed information on the UCLI commands, see the UCLI User Guide.

Functional Coverage

The VCS implementation of SystemVerilog supports the `covergroup` construct, which you specify as the user. These constructs allow the system to monitor values and transitions for variables and signals. They also enable cross coverage between variables and signals.

If you have covergroups in your design, VCS collects the coverage data during simulation and generates a database, `simv.vdb`. Once you have `simv.vdb`, you can use the Unified Report Generator to generate text or HTML reports. See “[Unified Report Generator \(URG\)](#)” on page 12-13.

Unified Coverage Directory and Database Control

A coverage directory named `simv.vdb` contains all the testbench functional coverage data. This is different from previous versions of VCS, where the coverage database files were stored, by default, in the current working directory or the path specified by `coverage_database_filename`. For your reference, VCS associates a logical test name with the coverage data that is generated by a simulation. VCS assigns a default test name, which you can override, by using the

`$coverage_set_test_database_name` task:

```
$coverage_set_test_database_name  
      ("test_name" [, "dir_name"] ) ;
```

VCS avoids overwriting existing database file names by generating unique non-clashing test names for consecutive tests.

For example, if the coverage data is to be saved to a test name called `pci_test`, and a database with that test name already exists in the coverage directory `simv.vdb`, then VCS automatically generates the new name `pci_test_gen1` for the next simulation run. The following table explains the unique name generation scheme details.

Table 14-1 Unique Name Generation Scheme

Test Name	Database for the...
<code>pci_test</code>	1st testbench run
<code>pci_test_gen_1</code>	2nd testbench run
<code>pci_test_gen_2</code>	3rd testbench run
<code>pci_test_gen_n</code>	n th testbench run

You can disable this method of ensuring database backup and force VCS to always overwrite an existing coverage database. To do this, use the following system task:

```
$coverage_backup_database_test (flag);
```

The value of flag can be:

- OFF for disabling database backup.
- ON for enabling database backup.

In order to not save the coverage data to a database file (for example, if there is a verification error), use the following system task:

```
$coverage_save_database (flag);
```

The value of flag can be:

- OFF for disabling database backup.
 - ON for enabling database backup.
-

Loading Coverage Data

Both cumulative coverage data and instance-specific coverage data can be loaded. The loading of coverage data from a previous VCS run implies that the bin hits from the previous VCS run to this run are to be added.

Loading Cumulative Coverage Data

The cumulative coverage data can be loaded either for all coverage groups, or for a specific coverage group. To load the cumulative coverage data for all coverage groups, use the following syntax:

```
$coverage_load_cumulative_data("test_name" [, "dir_name"]);
```

In this task, "dir_name" is optional. If you do not specify a "dir_name", by default, simv.vdb is taken as the directory containing the database.

The above task directs VCS to find the cumulative coverage data for all coverage groups found in the specified database file and to load this data if a coverage group with the appropriate name and definition exists in this VCS run.

To load the cumulative coverage data for just a single coverage group, use the following syntax:

```
$coverage_load_cumulative_cg_data("test_name",
                                    "covergroup_name" [, "dir_name"]);
```

In this task, "dir_name" is optional. If you do not specify a "dir_name", by default, simv.vdb is taken as the directory containing the database.

In [Example 14-1](#), there is a SystemVerilog class MyClass with an embedded covergroup covType. VCS finds the cumulative coverage data for the coverage group MyClass:covType in the database file Run1 and loads it into the covType embedded coverage group in MyClass.

Example 14-1

```
class MyClass;
    integer m_e;
    covergroup covType @m_e;
        cp1 : coverpoint m_e;
    endgroup
endclass
...
$coverage_load_cumulative_cg_data("Run1",
                                    "MyClass::covType") ;
```

Loading Instance Coverage Data

The coverage data can be loaded for a specific coverage instance. To load the coverage data for a standalone coverage instance, use the following syntax:

```
$covgLoadInstFromDbTest
    (coverage_instance, "test_name" [, "dir_name"] ) ;
```

In this task, "dir_name" is optional. If you do not specify a "dir_name", by default, simv.vdb is taken as the directory containing the database.

To load the coverage data for an embedded coverage instance, use the following syntax:

```
$covgLoadInstFromDbTest (class_object.cov_group_name,  
                         "test_name" [, "dir_name"] );
```

In this task, "dir_name" is optional. If you do not specify a "dir_name", by default, simv.vdb is taken as the directory containing the database.

The above commands direct VCS to find the coverage data for the specified instance name in the database, and load it into the coverage instance.

In [Example 14-2](#), there is a SystemVerilog class MyClass with an embedded covergroup covType. Two objects obj1 and obj2 are instantiated, each with the embedded covergroup covType. VCS will find the coverage information for the coverage instance obj1:covType from the database file Run1, and load this coverage data into the newly instantiated obj1 object. Note that the object obj2 will not be affected as part of this load operation.

Example 14-2

```
class MyClass;  
    integer m_e;  
    covergroup covType @m_e;  
        cp1 : coverpoint m_e;  
    endgroup  
endclass  
...  
MyClass obj1 = new;  
$covgLoadInstFromDbTest(obj1, "Run1");  
MyClass obj2 = new;
```

Note:

The compile-time or runtime options `-cm_dir` and `-cm_name` will overwrite the calls to `coverage_set_test_database_name` and load coverage data tasks.

`-cm_dir directory_path_name`

As a compile-time or runtime option, specifies an alternative name and location for the default `simv.vdb` directory. If not specified, VCS automatically adds the extension `.vdb` to the directory name. Note that if you compile the design with the `-cm_dir` option, and then move `simv.cm`, you must use `-cm_dir` at runtime to point to the new location of `simv.cm`.

`-cm_name filename`

As a compile-time or runtime option, specifies an alternative test name instead of the default name. The default test name is "test".

Using `-cov_disable_cg` to Disable Functional Coverage Items

The `-covg_disable_cg` option allows disabling all functional coverage items (covergroups) from the verification environment to isolate functional coverage issues or for performance reasons.

At times, it might be helpful to disable covergroups to analyze performance or zero in on specific issues with the verification environment. The `-covg_disable_cg` option can be passed at compile time or at runtime to eliminate the coverage factor.

When a design having covergroups shows up some issue (crashes/ performance related etc.), disabling all covergroups shows the problem still persist. If not, it is a functional coverage issue.

This option, when used with VCS at compile time, ignores all covergroups and their corresponding instances and references (loads/drivers/bidirects/xmrs). All statements that are dependent on covergroups will be impacted. (For example, if covergroup instances are referenced in statements like \$display/if/while etc., the whole statement is ignored.)

At runtime, -covg_disable_cg return from the top level calls to Coverage Library. Nothing relating to initializing/allocating/processing runs.

Memory Profiler

VCS supports profiling of memory consumed by the following dynamic data types using the system task `$vcsmemprof`:

- associative Array
- dynamic Array
- smart Queue
- string
- event
- class

The memory profiler reports the memory consumed by all the active instances of the different dynamic data types. Each invocation of `$vcsmemprof()` writes the profiler data in the user specified file. The time at which the call is made is also reported. This enables you to narrow down the search for any memory issues.

Syntax

The syntax for `$vcsmemprof()` is as follows:

```
$vcsmemprof ("filename", "w|a+");
```

Where:

`filename`

Name of the file where the memory profiler dumps the report.

w | a+

w and a+ designate the mode in which the file is opened. Specify w for writing and a+ for appending to an existing file.

Usage Model

\$vcsmemprof system task can be executed in the following ways:

- Within the module/program block
- From the UCLI prompt

Calling within the program is very straightforward as shown in the following example:

```
program test;
  ...
initial
  #5 $vcsmemprof ("my.prof", "w");
  ...
endprogram
```

However, if you want to call from the UCLI prompt, you need to use the UCLI command call to execute any system task from the UCLI prompt. For example:

```
% simv -ucli
ucli>call {$vcsmemprof("my.prof", "w")}
```

To use \$vcsmemprof during runtime, you should compile the design using the +dmpref option. The usage model is as follows:

Compilation

```
% vcs [compile_options] Verilog_files
```

Simulation

```
% simv [simv_options]
```

Memory Profile Report

The memory profiler reports only memory actively held at the current simulation time instant by the dynamic data types.

Consider the following program:

```
program automatic main;

task t1();
    integer arr1[];
    arr1 = new[ 500];
    #5;
endtask

task t2();
    integer arr2[];
    #5 arr2 = new[500];
    #5;
endtask

initial
begin
    fork
        begin t1(); end
        begin t2(); end
    join
end
endprogram
```

Note:

Once a scope is exited, variables local to that scope will be cleaned, and their memory will be reclaimed.

In this program, if `$vcsmemprof()` is called between 0 and 4 ns, then both `arr1` and `arr2` are active. If the call is made between 5 and 10 ns, then only `arr2` is active and after 10 ns, neither is active.

The profile report includes the following sections:

1. Top-level view

Reports the total dynamic memory consumed in all the programs (SV) and that consumed in all the modules in the system.

2. Module View

Reports the dynamic memory consumed in each SV module in the system.

3. Program View

Reports the dynamic memory consumed in each OV program in the system.

4. Program-To-Construct View

- Task-Function-Thread section

Reports the total dynamic memory in each active task, function and thread in the module/program.

- Class Section

Reports the total dynamic memory consumed in each class in the module/program.

- Dynamic data Section

Reports the total memory consumed in each of the dynamic testbench data types - associative arrays, dynamic arrays, queues, events, strings, in the module/program.

5. Module-To-Construct View

Same as "Program-To-Construct View".

Example 14-3

```
program automatic test;

  class FirstClass;
    integer b;
  endclass

  class SecondClass ;
    integer c;
    integer d[10];
  endclass

  task FirstTask() ;
    FirstClass a ;
    a = new;
    #100;
  endtask
```

```

task SecondTask() ;
    FirstClass a ;
    SecondClass b ;
    a = new;
    b = new;
    #100;
endtask

integer i;
integer sqProgram[$];
integer sqFork[$];

initial
begin
    #10;
    FirstTask();
    SecondTask();
    sqFork.push_front(1);
    sqProgram.push_front(1);
end
endprogram : test

```

Compilation

```
% vcs test.sv +dmprof -debug_all
```

Simulation

```
% simv -ucli
ucli> run 140
ucli> call {$vcsmemprof("memprof.txt", "w") }
ucli> run

//      Synopsys VCS Y-2006.06-SP1-9 Compiled Simulator
//      Memory taken by dynamic objects is reported
//          i.e class objects, dynamic arrays, assoc arrays
//          events, strings and smart queues
//      Memory is reported in bytes
=====
$vcsmemprof called at simulation time =           140
=====
=====
TOP LEVEL VIEW
=====
TYPE             MEMORY          %TOTALMEMORY
-----
MODULES          0              0.00
PROGRAMS        368            100.00
-----
=====
PROGRAM VIEW
=====
Program(index)      Memory      %TotalMemory    No
of Instances   Definition
-----
test_1(1)          368         100.00          1  test.sv:1.
-----
```

```
=====
PROGRAM TO CONSTRUCT MAPPING
=====

1. test_1

Class Data
-----
Name      Memory   %Total    #objects    #objects     Allocated at
          Memory      allocated      active
-----
SecondClass      128      34.78        1           1      test.sv:21
FirstClass       96       26.09        2           2      test.sv:20
                           test.sv:13
-----

Dynamic Data
-----
Type            Memory      %TotalMemory    #Alive Instances
-----
Queues          144        39.13          2
-----
```

Extensions to SystemVerilog

This section contains descriptions of Synopsys enhancements to IEEE-2005 SystemVerilog, in VCS release 2008.12 or later. This section contains the following topics:

- “Unique/Priority Case/IF Final Semantic Enhancements”

Unique/Priority Case/IF Final Semantic Enhancements

The behavior of the compliance checking keywords `unique` and `priority` for `case` and for `if...else if...else` selection statements as defined in the IEEE 1800-2005 LRM Section 10.4 in

some cases can cause spurious warnings when used inside a module's continuous assignment or always block. By default, VCS will evaluate compliance with unique or priority on every update to the selection statement input.

To force unique and priority to evaluate compliance only on the stable and final value of the selection input at the end of a simulation timestep, VCS now provides a compile time switch -xlrn uniq_prior_final.

This can be useful, for example, when always_comb might trigger several times within a simulation time slot while its input values are getting stabilized. The case statements can get executed several times during same time slot if it is valid for combinational blocks. While going through intermediate transitions, the case statement might get values that violate the unique or priority property and cause VCS to report multiple runtime warnings. When it is undesirable to receive intermediate warnings, compile time option '-xlrn uniq_prior_final' can be used to evaluate compliance for only the final stable value of the input.

Using Unique/Priority Case/If with Always Block or Continuous Assign

-xlrn uniq_prior_final behavior only applies to the use of unique and priority keywords when selection statements are used inside a module's continuous assignment or always block. The switch is not applicable for program block or initial block of code.

The following two examples illustrate this behavior:

Example 14-4 unique case statement at the same timestep

```
//test.sv:
```

```

module top;
reg cond;
bit [7:0] a = 0,b, v1, v2;
always_comb begin
    if (cond) begin
        unique case (a)
            v1: begin b = 0; $display(" Executing Case
                                with cond value 1 "); end
            v2: begin b = 1; $display(" Executing Case
                                with cond value 1 "); end
        endcase
    end
    else begin
        unique case (a)
            v1: begin b = 0; $display(" Executing Case
                                with cond value 0 "); end
            v2: begin b = 1; $display(" Executing Case
                                with cond value 0 "); end
        endcase
    end
end

initial begin
#1 cond = 1;
a=a+4; v1=4; v2=4;
$display("\n TIME %0d ns : cond value %0b, a value %0d",
        $time, cond, a);
#0 cond = 0;
a=a+1; v1++; v2++;
$display("\n TIME %0d ns: cond value %0b, a value %0d",
        $time, cond, a);
end
endmodule

```

Simulation output without '-x1rm uniq_prior_final':

```

%> vcs -sverilog test.sv -R

Executing Case with condition value 0
RT Warning: More than one conditions match in 'unique case'
statement.
"unique_case.sv", line 12, for top.

```

```

        Line    13 &    14 are overlapping at time      0.
Executing Case with cond value 0
RT Warning: More than one conditions match in 'unique case'
statement.

        "unique_case.sv", line 12, for top.
        Line    13 &    14 are overlapping at time      0.

TIME 1 ns : cond value 1, a value 4
Executing Case with cond value 1
RT Warning: More than one conditions match in 'unique case'
statement.

        "unique_case.sv", line 6, for top.
        Line    7 &    8 are overlapping at time      1.

TIME 1 ns: cond value 0, a value 5
Executing Case with cond value 0
RT Warning: More than one conditions match in 'unique case'
statement.

        "unique_case.sv", line 12, for top.
        Line    13 &    14 are overlapping at time      1.

```

Simulation output with ' -xlrm uniq_prior_final' compile time switch:

```

%> vcs -sverilog test.sv -xlrm uniq_prior_final -R
Executing Case with cond value 0:
RT Warning: More than one conditions match in 'unique case'
statement.

        "unique_case.sv", line 12, for top.
        Line    13 &    14 are overlapping at time      0.

TIME 1 ns : cond value 1, a value 4
Executing Case with cond value 1

TIME 1 ns: cond value 0, a value 5
Executing Case with cond value 0
RT Warning: More than one conditions match in 'unique case'
statement.

        "unique_case.sv", line 12, for top.
        Line    13 &    14 are overlapping at time      1.

```

Example 14-5 unique if inside always_comb

```
//test.sv
module top;
reg cond;
bit [7:0] a = 0,b;
always_comb begin

unique if (a == 0 || a == 1) $display ("A is 0 or 1");
else if (a == 2) $display ("A is 2");

end

initial begin
#100;
a = 1;
#100 a = 2;
#100 a = 3;
#0 a++;
#0 a++;
#0 a++;
#10 $finish;

end

endmodule
```

Simulation output without ‘-x1rm’:

```
%> vcs -sverilog test.sv -R

A is 0 or 1
A is 0 or 1
A is 0 or 1
A is 2
RT Warning: No condition matches in 'unique if' statement.
"unique_if.sv", line 5, for top, at time 300.
RT Warning: No condition matches in 'unique if' statement.
"unique_if.sv", line 5, for top, at time 300.
RT Warning: No condition matches in 'unique if' statement.
"unique_if.sv", line 5, for top, at time 300.
RT Warning: No condition matches in 'unique if' statement.
```

```
"unique_if.sv", line 5, for top, at time 300.  
$finish called from file "unique_if.sv", line 17.
```

Simulation output with '-xlrn uniq_prior_final':

```
%> vcs -sverilog test.sv -xlrn uniq_prior_final -R  
  
A is 0 or 1  
A is 0 or 1  
A is 0 or 1  
A is 2  
RT Warning: No condition matches in 'unique if' statement.  
"unique_if.sv", line 5, for top, at time 300.  
$finish called from file "unique_if.sv", line 17.
```

Using Unique/Priority Inside a Function

With the new enhancement, if unique/priority case statement is used inside a function, VCS not only points to the current case statement but also provides a complete stack trace of where the function is called. The following example illustrate this behavior:

Example 14-6 unique case used with nested loop inside function

```
//test.sv  
module top;  
    int i,j;  
    reg [1:0] [2:0] a, b, c;  
    bit flag;  
  
    function foo;  
        for (int i=0; i<2; i++)  
            for (int j=0; j<3; j++)  
                unique case (a[i][j])  
                    0: b[i][j] = 1'b0;  
                    1: b[i][j] = c[i][j];  
                endcase  
    endfunction : foo
```

```

    always_comb begin
        for(i=0; i<4; i++) begin
            if (i==2)
                foo();
        end
    end

    initial begin
        a = 6'b00x011;
    end

endmodule : top

```

Simulation output without '-xlr' option:

```
%> vcs -sverilog test.sv -R

RT Warning: No condition matches in 'unique case' statement.
"unique_case_inside_func.sv", line 8, for top.foo, at time 0.

RT Warning: No condition matches in 'unique case' statement.
"unique_case_inside_func.sv", line 8, for top.foo, at time 0.
```

Simulation output with '-xlr' uniq_prior_final':

```
%> vcs -sverilog test.sv -xlr uniq_prior_final -R

RT Warning: No condition matches in 'unique case' statement.

"unique_case_inside_func.sv", line 8, for top.foo, at time 0.
#0 in foo      at unique_case_inside_func.sv:8
#1 in loop with j= 0      at unique_case_inside_func.sv:7
#2 in loop with i= 1      at unique_case_inside_func.sv:6
#3 in top      at unique_case_inside_func.sv:16
#4 in loop with i= 2      at unique_case_inside_func.sv:14
```

Note:

The following limitations must be noted while using '-xlr' `uniq_prior_final`' feature for loop indices:

- It must be written in `for` statement. The `while` and `do...while` are not supported.
- The loop bounds must be compile-time constants.
 - `for(i= lsb; i<msb; i++)`
 - Here, `lsb` and `msb` must be compile-time constant, or will become constant when upper loops get unrolled.
 - No other drivers of the loop variable must be in the loop body.

VCS also supports `unique/prior final` in a `for` loop that can not be unrolled at compile time. For example, if you have a `for` loop whose range could not be determined at compile-time and if there are errors during the last evaluation of such a `for` loop, VCS still reports the error. However, loop index information will not be provided. Even if multiple failures occur in different iterations, VCS reports only the last one.

Important:

Use `unique/priority case/if` statement only inside `always` block, `continuous assign`, or inside a function. If you use it in other places, the `final` semantic will be ignored.

System Tasks to Control Warning Messages

Two system tasks `$uniq_prior_checkon` and `$uniq_prior_checkoff` will enable you to switch on/off runtime warning messages for `unique/priority if/case` statements. The following example illustrates the use model of these tasks to ignore violations:

Example 14-7 System tasks to control warning messages

```
//test.sv
module m;
```

```

bit sel, v1, v2;

//Disable this initial block to display all RT warning
messages
initial
begin
    $display($time, " Priority checker OFF\n");
    $uniq_prior_checkoff();
    #1;
    $display($time, " Priority checker ON\n");
    $uniq_prior_checkon();
end

initial
begin
    //violation with this set of values (warning disabled)
    sel = 1'b1;
    v1 = 1'b1;
    v2 = 1'b1;
    #1;
    //violation with this set of values (warning enabled)
    sel = 1'b0;
    v1 = 1'b0;
    v2 = 1'b0;
    #1;
end
always_comb begin
unique case(sel)
    v1: $display($time, " Hello");
    v2: $display($time, " World");
endcase
end
endmodule

```

Simulation Output:

```
%> vcs -sverilog test.sv -R
```

```

0 Priority checker OFF
0 Hello
0 Hello
1 Priority checker ON

```

```
1 Hello
RT Warning: More than one conditions match in 'unique case'
statement.
"system_task_control_warning.sv", line 28, for m.
Line 29 & 30 are overlapping at time 1.
```


15

Aspect Oriented Extensions

The Aspect oriented extensions is a Limited Customer availability (LCA) feature in NTB(SV) and requires a separate license. Please contact your Synopsys AC for a license key.

Aspect-Oriented Programming (AOP) methodology complements the OOP methodology using a construct called aspect or an aspect-oriented extension (AOE) that can affect the behavior of a class or multiple classes. In AOP methodology, the terms “aspect” and “aspect-oriented extension” are used interchangeably.

Aspect oriented extensions in SV allow testbench engineers to design testcase more efficiently, using fewer lines of code.

AOP addresses issues or concerns that prove difficult to solve when using Object-Oriented Programming (OOP) tow write constrained-random testbenches.

Such concerns include:

1. Context-sensitive behavior.
2. Unanticipated extensions.
3. Multi-object protocols.

In AOP these issues are termed cross-cutting concerns as they cut across the typical divisions of responsibility in a given programming model.

In OOP, the natural unit of modularity is the class. Some of the cross cutting concerns, such as "Multi-object protocols" , cut across multiple classes and are not easy to solve using the OOP methodology. AOP is a way of modularizing such cross-cutting concerns. AOP extends the functionality of existing OOP derived classes and uses the notion of aspect as a natural unit of modularity. Behavior that affects multiple classes can be encapsulated in aspects to form reusable modules. As potential benefits of AOP are achieved better in a language where an aspect unit can affect behavior of multiple classes and therefore can modularize the behavior that affects multiple classes, AOP ability in SV language is currently limited in the sense that an aspect extension affects the behavior of only a single class. It is useful nonetheless, enabling test engineers to design code that efficiently addresses concerns "Context-sensitive behavior" and "Unanticipated extensions".

AOP is used in conjunction with object-oriented programming. By compartmentalizing code containing aspects, cross-cutting concerns become easy to deal with. Aspects of a system can be changed, inserted or removed at compile time, and become reusable.

It is important to understand that the overall verification environment should be assembled using OOP to retain encapsulation and protection. NTB's Aspect-Oriented Extensions should be used only for constrained-random test specifications with the aim of minimizing code.

SV's Aspect-Oriented Extensions should not be used to:

- Code base classes and class libraries
- Debug, trace or monitor unknown or inaccessible classes
- Insert new code to fix an existing problem

For information on the creation and refinement of verification testbenches, see the *Reference Verification Methodology User Guide*.

Aspect-Oriented Extensions in SV

In SV, AOP is supported by a set of directives and constructs that need to be processed before compilation. Therefore, an SV program with these Aspect oriented directives and constructs would need to be processed as per the definition of these directives and constructs in SV to generate an equivalent SV program that is devoid of aspect extensions, and consists of traditional SV. Conceptually, AOP is implemented as pre-compilation expansion of code.

This chapter explains how AOE in SV are directives to SV compiler as to how the pre-compilation expansion of code needs to be performed.

In SV, an aspect extension for a class can be defined in any scope where the class is visible, except for within another aspect extension. That is, aspect extensions can not be nested.

An aspect oriented extension in SV is defined using a new top-level *extends directive*. Terms aspect and “extends directive” have been used interchangeably throughout the document. Normally, a class is extended through derivation, but an extends directive defines modifications to a pre-existing class by doing *in-place* extension of the class. *in-place* extension modifies the definition of a class by adding new member fields and member methods, and changing the behavior of earlier defined class methods, without creating any new subclass(es). That is, SV’s Aspect-Oriented Extensions change the original class definition without creating subclasses. These changes affect all instances of the original class that was extended by AOEs.

An extends directive for a class defines a scope in SV language. Within this scope exist the items that modify the class definition. These items within an extends directive for a class can be divided into the following three categories.

- **Introduction**

Declaration of a new property, or the definition of a new method, a new constraint, or a new coverage group within the extends directive scope adds (or *introduces*) the new symbol into the original class definition as a new member. Such declaration/definition is called an *introduction*.

- **Advice**

An *advice* is a construct to specify code that affects the behavior of a member method of the class by *weaving* the specified code into the member method definition. This is explained in more detail later. The advice item is said to be an advice *to* the affected member method.

- Hide list:

Some items within an extends directive, such as a virtual method introduction, or an advice to virtual method may not be permissible within the extends directive scope depending upon the *hide permissions* at the place where the item is defined. A *hide list* is a construct whose placement and arguments within the extends directive scope controls the hide permissions. There could be multiple hide lists within an extends directive.

Processing of AOE as a Precompilation Expansion

As a precompilation expansion, AOE code is processed by VCS to modify the class definitions that it extends as per the directives in AOE.

A *symbol* is a valid identifier in a program. Classes and class methods are symbols that can be affected by AOE. AOE code is processed which involves adding of introductions and *weaving* of advices in and around the affected symbols. Weaving is performed before actual compilation (and thereby before symbol resolution), therefore, under certain conditions, introduced symbols with the same identifier as some already visible symbol, can *hide* the already visible symbols. This is explained in more detail in [Section , “hide_list details,” on page 15-31](#). The preprocessed input program, now devoid of AOE, is then compiled.

Syntax:

```
extends_directive ::=  
    extends extends_identifier  
(class_identifier) [dominate_list];  
        extends_item_list  
    endextends  
  
dominate_list ::=  
    dominates(extends_identifier  
{,extends_identifier});  
  
extends_item_list ::=  
    extends_item {extends_item}  
  
extends_item ::=  
    class_item  
    | advice  
    | hide_list  
  
class_item ::=  
    class_property  
    | class_method  
    | class_constraint  
    | class_coverage  
    | enum_defn  
  
advice ::= placement procedure  
  
placement ::=  
    before  
    | after  
    | around  
  
procedure ::=  
    | optional_method_specifiers task  
        task_identifier(list_of_task_proto_formals);  
    | optional_method_specifiers function  
function_type  
  
function_identifier(list_of_function_proto_formals)  
endfunction  
  
advice_code ::= [stmt] {stmt}
```

```

stmt ::= statement
      | proceed ;

hide_list ::= 
    hide([hide_item {, hide_item}]) ;

hide_item ::=
    // Empty
    | virtuals
    | rules

```

The symbols in boldface are keywords and their syntax are as follows:

extends_identifier

Name of the aspect extension.

class_identifier

Name of the class that is being extended by the extends directive.

dominate_list

Specifies extensions that are *dominated* by the current directive. Domination defines the *precedence* between code woven by multiple extensions into the same scope. One extension can dominate one or more of the other extensions. In such a case, you must use a comma-separated list of extends identifiers.

```

dominates(extends_identifier
{, extends_identifier} );

```

A dominated extension is assigned lower precedence than an extension that dominates it. Precedence among aspects extensions of a class determines the order in which introductions defined in the

aspects are added to the class definition. It also determines the order in which advices defined in the aspects are *woven* into the class method definitions thus affecting the behavior of a class method. Rules for determination of precedence among aspects are explained later in “[Precedence](#)” on page 15-17.

class_property

Refers to an item that can be parsed as a property of a class.

class_method

Refers to an item that can be parsed as a class method.

class_constraint

Refers to an item that can be parsed as a class constraint.

class_coverage

Refers to an item that can be parsed as a coverage_group in a class.

advice_code

Specifies to a block of statements.

statement

Is an SV statement.

procedure_prototype

A full prototype of the target procedure. Prototypes enable the advice code to reference the formal arguments of the procedure.

opt_method_specifiers

Refers to a combination of protection level specifier (**local**, or **protected**), virtual method specifier (**virtual**), and the static method specifier (**static**) for the method.

task_identifier

Name of the task.

function_identifier

Name of the function.

function_type

Data type of the return value of the function.

list_of_task_proto_formals

List of formal arguments to the task.

list_of_function_proto_formals

List of formal arguments to the function.

placement

Specifies the position at which the advice code within the advice is *woven* into the *target method* definition. Target method is either the class method, or some other new method that was created as part of the process of *weaving*, which is a part of pre-compilation expansion of code. The overall details of the process of “weaving” are explained in [Pre-compilation Expansion details](#). The placement element could

be any of the keywords, **before**, **after**, or **around**, and the advices with these placement elements are referred to as *before advice*, *after advice* and *around advice*, respectively.

proceed statement

The proceed keyword specifies an SV statement that can be used within advice code. A proceed statement is valid only within an around block and only a single proceed statement can be used inside the *advice code block* of an around advice. It cannot be used in a before advice block or an after advice block. The proceed statement is optional.

hide_list

Specifies the permission(s) for introductions to hide a symbol, and/or permission(s) for advices to modify local and protected methods. It is explained in detail in [Section , “hide_list details,” on page 15-31.](#)

Weaving advice into the target method

The target method is either the class method, or some other new method that was created as part of the process of *weaving*. “Weaving” of all advices in the input program comprises several steps of *weaving of an advice into the target method*. Weaving of an advice into its target method involves the following.

A new method is created with the same method prototype as the target method and with the advice code block as the code block of the new method. This method is referred to as the *advice method*.

The following table shows the rest of the steps involved in weaving of the advice for each type of placement element (**before**, **after**, and **around**).

Table 15-1 Placement Elements

Element	Description
before	Inserts a new method-call statement that calls an advice method. The statement is inserted as the first statement to be executed before any other statements.
after	Creates a new method A with the target method prototype, with its first statement being a call to the target method. Second statement with A is a new method call statement that calls the advice method. All the instances in the input program where the target method is called are replaced by newly created method calls to A. A is replaced as the new target method.
around	All the instances in the input program where the target method is called are replaced by newly created method calls to the advice method.

Within an extends directive, you can specify only one advice can be specified for a given placement element and a given method. For example, an extends directive may contain a maximum of one before, one after, and one around advice each for a class method *Packet::foo* of a class *Packet*, but it may not contain two before advices for the *Packet::foo*.

Target method:

```
task myTask();
    $display("Executing original code\n");
endtask
```

Advice:

```
before task myTask ();
    $display("Before in aoel\n");
endtask
```

Weaving of the advice in the target method yields the following.

```
task myTask();
    mytask_before();
    $display("Executing original code\n");
endtask

task mytask_before();
    $display("Before in aoel\n");
endtask
```

Note that the SV language does not impose any restrictions on the names of newly created methods during pre-compilation expansion, such as *mytask_before*. Compilers can adopt any naming conventions such methods that are created as a result of the *weaving* process.

Example 15-1

Target method:

```
task myTask();
    $display("Executing original code\n");
endtask
```

Advice:

```
after task myTask ();
    $display("Before in aoel\n");
endtask
```

Weaving of the advice in the target method yields the following.

```
task myTask_newTarget() ;
    myTask();
    myTask_after();
endtask

task myTask();
    $display("Executing original code\n");
endtask

task myTask_after();
    $display("After in aoe1\n");
endtask
```

As a result of weaving, all the method calls to myTask() in the input program code are replaced by method calls to myTask_newTarget(). Also, myTask_newTarget replaces myTask as the target method for myTask().

Target method:

```
task myTask();
    $display("Executing original code\n");
endtask
```

Advice:

```
around task myTask ();
    $display("Around in aoe1\n");
endtask
```

Weaving of the advice in the target method yields the following.

```
task myTask_around();
    $display("Around in aoel\n");
endtask

task myTask();
    $display("Executing original code\n");
endtask
```

As a result of weaving, all the method calls to myTask() in the input program code are replaced by method calls to myTask_around(). Also, myTask_around() replaces myTask() as the target method for myTask().

During weaving of an **around** advice that contains a **proceed** statement, the **proceed** statement is replaced by a method call to the target method.

Example 15-2

Target method:

```
task myTask();
    $display("Executing original code\n");
endtask
```

Advice:

```
around task myTask ();
    proceed;
    $display("Around in aoel\n");
endtask
```

Weaving of the advice in the target method yields:

```
task myTask_around();
    myTask();
    $display("Around in aoe1\n");
endtask

task myTask();
    $display("Executing original code\n");
endtask
```

As a result of weaving, all the method calls to `myTask()` in the input program code are replaced by method calls to `myTask_around()`. The proceed statement in the around code is replaced with a call to the target method `myTask()`. Also, `myTask_around` replaces `myTask` as the target method for `myTask()`.

Pre-compilation Expansion details

Pre-compilation expansion of a program containing AOE code is done in the following order:

1. Preprocessing and parsing of all input code.
2. Identification of the symbols, such as methods and classes affected by extensions.
3. The precedence order of aspect extensions (and thereby introductions and advices) for each class is established.
4. Addition of introductions to their respective classes as class members in their order of precedence. Whether an introduction can or can not override or hide a symbol with the same name that is visible in the scope of the original class definition, is dependent on certain rules related to the `hide_list` parameter. For a detailed explanation, see “[“`hide_list` details” on page 15-31](#)”.

5. Weaving of all advices in the input program are weaved into their respective class methods as per the precedence order.

These steps are described in more detail in the following sections.

Precedence

Precedence is specified through the *dominate_list* (see “*dominate_list*” on page 15-8) There is no default precedence across files; if precedence is not specified, the tool is free to weave code in any order. Within a file, dominance established by *dominate_lists* always overrides precedence established by the order in which *extends* directives are coded. Only when the precedence is not established after analyzing the dominate lists of directives, is the order of coding used to define the order of precedence.

Within an *extends* directive there is an inherent precedence between advices. Advices that are defined later in the directive have higher precedence than those defined earlier.

Precedence does not change the order between adding of introductions and weaving of advices in the code. Precedence defines the order in which introductions to a class are added to the class, and the order in which advices to methods belonging to a class are woven into the class methods.

Example 15-3

```
// Beginning of file Input.vr

program top ;
    initial begin
        packet p;
        p = new();
        p.send();
    end
endprogram
```

```

class packet;
...
// Other member fields/methods
...
task send();
    $display("Sending data\n");
endtask
endclass

extends aspect_1(packet) dominates (aspect_2, aspect_3);
    after task send();                                // Advice 1
        $display("Aspect_1: send advice after\n");
    endtask
endextends

extends aspect_2(packet);
    after task send() ;                            // Advice 2
        $display("Aspect_2: send advice after\n");
    endtask
endextends

extends aspect_3(packet);
    around task send();                           // Advice 3
        $display("Aspect_3: Begin send advice around\n");
        proceed;
        $display("Aspect_3: End send advice around\n");
    endtask
    before task send();                          // Advice 4
        $display("Aspect_3: send advice before\n");
    endtask
endextends

// End of file Input.vr

```

In Example 15-3, multiple aspect extensions for a class named *packet* are defined in a single SV file. As specified in the dominating list of *aspect_1*, *aspect_1* dominates both *aspect_2* and *aspect_3*. As per the dominating lists of the aspect extensions, there is no precedence order established between *aspect_2* and *aspect_3*, and

since `aspect_3` is coded later in `Input.vr` than `aspect_2`, `aspect_3` has higher precedence than `aspect_2`. Therefore, the precedence of these aspect extensions in the decreasing order of precedence is:

{`aspect_1`, `aspect_3`, `aspect_2`}

This implies that the advice(s) within `aspect_2` have lower precedence than advice(s) within `aspect_3`, and advice(s) within `aspect_3` have lower precedence than advice(s) within `aspect_1`. Therefore, `advice 2` has lower precedence than `advice 3` and `advice 4`. Both `advice 3` and `advice 4` have lower precedence than `advice 1`. Between `advice 3` and `advice 4`, `advice 4` has higher precedence as it is defined later than `advice 3`. That puts the order of advices in the increasing order of precedence as:

{2, 3, 4, 1}.

Adding of Introductions

Target scope refers to the scope of the class definition that is being extended by an aspect. Introductions in an aspect are appended as new members at the end of its target scope. If an extension A has precedence over extension B, the symbols introduced by A are appended first.

Within an aspect extension, symbols introduced by the extension are appended to the target scope in the order they appear in the extension.

There are certain rules according to which an introduction symbol with the same identifier name as a symbol that is visible in the target scope, may or may not be allowed as an introduction. These rules are discussed later in the chapter.

Weaving of advices

An input program may contain several aspect extensions for any or each of the different class definitions in the program. Weaving of advices needs to be carried out for each class method for which an advice is specified.

Weaving of advices in the input program consists of weaving of advices into each such class method. Weaving of advices into a class method A is unrelated to weaving of advices into a different class method B, and therefore weaving of advices to various class methods can be done in any ordering of the class methods.

For weaving of advices into a class method, all the advices pertaining to the class method are identified and ordered in the order of increasing precedence in a list L. This is the order in which these advices are woven into the class method thereby affecting the run-time behavior of the method. The advices in list L are woven in the class method as per the following steps. Target method is initialized to the class method.

- a. Advice A that has the lowest precedence in L is woven into the target method as explained earlier. Note that the target method may either be the class method or some other method newly created during the weaving process.
- b. Advice A is deleted from list L.
- c. The next advice on list L is woven into the target method. This continues until all the advices on the list have been woven into list L.

It would become apparent from the example provided later in this section how the order of precedence of advices for a class method affects how advices are woven into their target method and thus the relative order of execution of advice code blocks. Before and after advices within an aspect to a target method are unrelated to each

other in the sense that their relative precedence to each other does not affect their relative order of execution when a method call to the target method is executed. The before advice's code block executes before the target method code block, and the after advice code block executes after the target method code block. When an around advice is used with a before or after advice in the same aspect, code weaving depends upon their precedence with respect to each other. Depending upon the precedence of the around advice with respect to other advices in the aspect for the same target method, the around advice either may be woven before all or some of the other advices, or may be woven after all of the other advices.

As an example, weaving of advices 1, 2, 3, 4 specified in aspect extensions in Example 15-3 leads to the expansion of code in the following manner. Advices are woven in the order of increasing precedence {2, 3, 4, 1} as explained earlier.

Example 15-4

```
// Beginning of file Input.vr

program top ;
    packet p;
    p = new();
    p.send_Created_a();
endprogram

class packet;
    ...
    // Other member fields/methods
    ...
    task send();
        p$display("Sending data\n");
    endtask

    task send_Created_a();
        send();
        send_after_Created_b();
    endtask

    task send_after_Created_b();

```

```

        $display("Aspect_2: send advice after\n");
    endtask

endclass

extends aspect_1(packet) dominates (aspect_2, aspect_3);
    after task send();                                // Advice 1
        $display("Aspect_1: send advice after\n");
    endtask
endextends

extends aspect_3(packet);
    around task send();                            // Advice 3
        $display("Aspect_3: Begin send advice around\n");
        proceed;
        $display("Aspect_3: End send advice around\n");
    endtask

    before task send();                           // Advice 4
        $display("Aspect_3: send advice before\n");
    endtask
endextends

// End of file Input.sv

```

This Example 15-4 shows what the input program looks like after weaving advice 2 into the class method. Two new methods *send_Created_a* and *send_after_Created_b* are created in the process and the instances of method call to the target method *packet::send* are modified, such that the code block from advice 2 executes after the code block of the target method *packet::send*.

Example 15-5

```

// Beginning of file Input.vr

program top;
    packet p;
    p = new();
    p.send_around_Created_c();
endprogram

class packet;
    ...
    // Other member fields/methods

```

```

    ...
task send();
    $display("Sending data\n");
endtask

task send_Created_a();
    send();
    send_after_Created_b();
endtask

task send_after_Created_b();
    $display("Aspect_2: send advice after\n");
endtask

task send_around_Created_c();
    $display("Aspect_3: Begin send advice around\n");
    send_Created_a();
    $display("Aspect_3: End send advice around\n");
endtask
endclass

extends aspect_1(packet) dominates (aspect_2, aspect_3);
    after task send();                                // Advice 1
        $display("Aspect_1: send advice after\n");
    endtask
endextends

extends aspect_3(packet);
    before task send();                             // Advice 4
        $display("Aspect_3: send advice before\n");
    endtask
endextends

// End of file Input.sv

```

This Example 15-5 shows what the input program looks like after weaving advice 3 into the class method. A new method *send_around_Created_c* is created in this step and the instances of method call to the target method *packet::send_Created_a* are modified, such that the code block from *advice 3* executes *around* the code block of method *packet::send_Created_a*. Also note that the *proceed* statement from the advice code block is replaced by a

call to `send_Created_a`. At the end of this step, `send_around_Created_c` becomes the new target method for weaving of further advices to `packet::send`.

Example 15-6

```
// Beginning of file Input.vr

program top;
    packet p;
    p = new();
    p.send_around_Created_c();
endprogram

class packet;
    ...
    // Other member fields/methods
    ...
    task send();
        $display("Sending data\n");
    endtask

    task send_Created_a();
        send();
        send_after_Created_b();
    endtask

    task send_after_Created_b();
        $display("Aspect_2: send advice after\n");
    endtask

    task send_around_Created_c();
        send_before_Created_d();
        $display("Aspect_3: Begin send advice around\n");
        send_after_Created_a();
        $display("Aspect_3: End send advice around\n");
    endtask

    task send_before_Created_d();
        $display("Aspect_3: send advice before\n");
    endtask
endclass

extends aspect_1(packet) dominates (aspect_2, aspect_3);
after task send();                                // Advice 1
$display("Aspect_1: send advice after\n");
```

```

        endtask
endextends

// End of file Input.sv

```

This Example 15-6 shows what the input program looks like after weaving advice 4 into the class method. A new method *send_before_Created_d* is created in this step and a call to it is added as the first statement in the target method *packet::send_around_Created_c*. Also note that the outcome would have been different if *advice 4* (before advice) was defined earlier than *advice 3* (around advice) within *aspect_3*, as that would have affected the order of precedence of *advice 3* and *advice*. In that scenario the *advice 3* (around advice) would have weaved around the code block from *advice 4* (before advice), unlike the current outcome.

Example 15-7

```

// Beginning of file Input.vr

program top;
    packet p;
    p = new();
    p.send_Created_f();
endprogram

class packet;
    ...
    // Other member fields/methods
    ...
    task send();
        $display("Sending data\n");
    endtask

    task send_Created_a();
        send();
        send_Created_b();
    endtask

    task send_after_Created_b();

```

```

        $display("Aspect_2: send advice after\n");
endtask

task send_around_Created_c();
    send_before_Created_d();
    $display("Aspect_3: Begin send advice around\n");
    send_after_Created_a();
    $display("Aspect_3: End send advice around\n");
endtask

task send_before_Created_d();
    $display("Aspect_3: send advice before\n");
endtask
task send_after_Created_e();
    $display("Aspect_1: send advice after\n");
endtask

task send_Created_f();
    send_around_Created_c();
    send_after_Created_e()
endtask
endclass

// End of file Input.sv

```

This Example 15-7 shows the input program after weaving of all four advices {2, 3, 4, 1}. New methods *send_after_Created_e* and *send_Created_f* are created in the last step of weaving and the instances of method call to *packet::send_around_Created_c* were replaced by method call to *packet::send_Created_f*.

When executed, output of this program is:

```

Aspect_3: send advice before
Aspect_3: Begin send advice around
Sending data
Aspect_2: send advice after
Aspect_3: End send advice around
Aspect_1: send advice after

```

Examples of code containing around advice

```

// Begin file input.vr

program top;
    foo f;
    f = new();
    f.myTask();
endprogram

class foo;
    int i;
    task myTask();
        $display("Executing original code\n");
    endtask
endclass
extends aoe1 (foo) dominates (aoe2);
    around task myTask();
        proceed;
        $display("around in aoe1\n");
    endtask
endextends
extends aoe2 (foo);
    around task myTask();
        proceed;
        $display("around in aoe2\n");
    endtask
endextends
// End file input.sv

```

When **aoe1** dominates **aoe2**, as in func1, the output when the program is executed is:

```

Executing original code
around in aoe2
around in aoe1

```

Example 15-8

```

// Begin file input.vr

program top;
    foo f;

```

```

        f = new();
        f.myTask();
    endprogram

    class foo;
        int i;
        task myTask();
            printf("Executing original code\n");
        endtask
    endclass
    extends aoe1 (foo);
        around task myTask();
            proceed;
            printf("around in aoe1\n");
        endtask
    endextends
    extends aoe2 (foo) dominates (aoe1);
        around task myTask();
            proceed;
            printf("around in aoe2\n");
        endtask
    endextends
// End file input.sv

```

On the other hand, when `aoe2` dominates `aoe1` as in this Example 15-8, the output is:

```

Executing original code
around in aoe1
around in aoe2

```

Symbol resolution details:

As introductions and advices defined within `extends` directives are pre-processed as a pre-compilation expansion of the input program, the pre-processing occurs earlier than final symbol resolution stage within a compiler. Therefore, it is possible for AOE code to reference symbols that were added to the original class definition using AOEs.

Because advices are woven after introductions have been added to the class definitions, advices can be specified for introduced member methods and can reference introduced symbols.

An advice to a class method can access and modify the member fields and methods of the class object to which the class method belongs. An advice to a class function can access and modify the variable that stores the return value of the function.

Furthermore, members of the original class definition can also reference symbols introduced by aspect extensions using the extern declarations (?). Extern declarations can also be used to reference symbols introduced by an aspect extension to a class in some other aspect extension code that extends the same class.

An introduction that has the same identifier as a symbol that is already defined in the target scope as a member property or member method is not permitted.

Examples:

Example 15-9

```
// Begin file example.vr

program top;
    packet p;
    p = new();
    p.foo();
endprogram

class packet;
    task foo(integer x); //Formal argument is "x"
        $display("x=%0d\n", x);
    endtask
endclass

extends myaspect(packet);
    // Make packet::foo always print: "x=99"
```

```

        before task foo(integer x);
            x = 99;      //force every call to foo to use x=99
        endtask
endextends

// End file example.sv

```

The extends directive in Example 15-9 sets the *x* parameter inside the *foo()* task to 99 before the original code inside of *foo()* executes. Actual argument to *foo()* is not affected, and is not set unless passed-by-reference using ref.

Example 15-10

```

// Begin file example.sv
program top ;
    packet p;
    p = new();
    $display("Output is: %d\n", p.bar());
endprogram

class packet ;
    function integer bar();
        bar = 5;
        $display("Point 1: Value = %d\n", bar);
    endfunction
endclass

extends myaspect(packet);
    after function integer bar();
        $display("Point 2: Value = %d\n", bar);
        bar = bar + 1;                      // Stmt A
        $display("Point 3: Value = %d\n", bar);
    endfunction
endextends

// End file example.sv

```

An advice to a function can access and modify the variable that stores the return value of the function as shown in Example 15-10, in this example a call to *packet::bar* returns 6 instead of 5 as the final return value is set by the *Stmt A* in the advice code block.

When executed, the output of the program code is:

```
Point 1: Value = 5
Point 2: Value = 5
Point 3: Value = 6
Output is: 6
```

hide_list details

The *hide_list* item of an *extends_directive* specifies the permission(s) for introductions to hide symbols, and/or advice to modify local and protected methods. By default, an introduction does not have permission to hide symbols that were previously visible in the target scope, and it is an error for an extension to introduce a symbol that hides a global or super-class symbol.

The *hide_list* option contains a comma-separated list of options such as:

- The **virtuals** option permits the hiding (that is, overriding) of virtual methods defined in a super class. Virtual methods are the only symbols that may be hidden; global, and file-local tasks and functions may not be hidden. Furthermore, all introduced methods must have the same virtual modifier as their overridden super-class and overriding sub-class methods.
- The **rules** option permits the extension to suspend access rules and to specify advice that changes protected and local virtual methods; by default, extensions cannot change protected and local virtual methods.
- An empty option list removes all permissions, that is, it resets permissions to default.

In Example 15-11, the *print* method introduced by the *extends* directive hides the *print* method in the super class.

Example 15-11

```
class pbase;
    virtual task print();
        $display("I'm pbase\n");
    endtask
endclass

class packet extends pbase;
    task foo();
        $display(); //Call the print task
    endtask
endclass

extends myaspect(packet);
    hide(virtuals); // Allows permissions to
                     // hide pbase::print task

    virtual task print();
        $display("I'm packet\n");
    endtask
endextends
```

As explained earlier, there are two types of hide permissions:

- a. Permission to hide virtual methods defined in a super class (option *virtuals*) is referred to as *virtuals-permission*. An *aspect item* is either an introduction, an advice, or a hide list within an aspect. If at an aspect item within an aspect, such permission is granted, then the *virtuals-permission* is said to be *on* or the *status* of *virtuals-permission* is said to be on at that aspect item and at all the aspect items following that, until a hide list that forfeits the permission is encountered. If *virtuals-permission* is not on or the *status* of *virtuals-permission* is not on at an aspect item, then the *virtuals-permission* at that item is said to be *off* or the *status* of *virtuals-permission* at that item is said to be *off*

- b. Permission to suspend access rules and to specify advice that changes protected and local virtual methods (option "rules") is referred to as *rules-permission*. If within an aspect, at an aspect item, such permission is granted, then the rules-permission is said to be *on* or the *status* of rules-permission is said to be *on* at that aspect item and at all the aspect items following that, until a hide list that forfeits the permission is encountered. If rules-permission is not on or the status of rules-permission is not on at an aspect item, then the rules-permission at that item is said to be *off* or the *status* of rules-permission at that item is said to be *off*.

Permission for one of the above types of hide permissions does not affect the other. Status of rules-permission and hide-permission varies with the position of an aspect item within the aspect. Multiple hide_list(s) may appear in the extension. In an aspect, whether an introduction or an advice that can be affected by hide permissions is permitted to be defined at a given position within the aspect extension is determined by the status of the relevant hide permission at the position. A hide_list at a given position in an aspect can change the status of rules-permission and/or virtuals-permission at that position and all following aspect items until any hide permission status is changed again in that aspect using hide_list.

Example 15-12 illustrates how the two hide permissions can change at different aspect items within an aspect extension.

Example 15-12

```

class pbase;
    virtual task print1();
        $display("pbase::print1\n");
    endtask

    virtual task print2();
        $display("pbase::print2\n");
    endtask
endclass

```

```

class packet extends pbase;
    task foo();
        $display();
    endtask

    local virtual task rules-test();
        $display("Rules-permission example\n");
    endtask
endclass

extends myaspect(packet);

// At this point within the myaspect scope,
// virtuals-permission and rules-permission are both off.

hide(virtuals); // Grants virtuals-permission

// virtuals-permission is on at this point within aspect,
// and therefore can define print1 method introduction.
virtual task print1();
    $display("packet::print1\n");
endtask

hide(); // virtuals-permission is forfeited

hide(rules); // Grants rules-permission

// Following advice permitted as rules-permission is on
before local virtual task rules-test();
    $display("Advice to Rules-permission example\n");
endtask

hide(virtuals); // Grants virtuals-permission

// virtuals-permission is on at this point within aspect,
// and therefore can define print2 method introduction.
virtual task print2();
    $display("packet::print2\n");
endtask
endextends

```

Examples

Introducing new members into a class:

Example 15-13 shows how AOE can be used to introduce new members into a class definition. *myaspect* adds a new property, constraint, coverage group, and method to the *packet* class.

Example 15-13

```
class packet;
    rand bit[31:0] ...
    ...
endclass

extends myaspect(packet);
    integer sending_port;

constraint con2 {
    hdr_len == 4;
}

coverage_group cov2 @(posedge CLOCK);
    coverpoint sending_port;
endgroup

task print_sender();
    $display("Sending port = %0d\n", sending_port);
endtask
endextends
```

As mentioned earlier, new members that are introduced should not have the same name as a symbol that is already defined in the class scope. So, AOE defined in the manner shown in Example 15-14 will not be allowed, as the aspect *myaspect* defines *x* as one of the introductions when the symbol *x* is already defined in class *foo*.

Example 15-14 : Non permissible introduction

```
class foo;
    rand integer myfield;
    integer x;
    ...
endclass

extends myaspect(foo);
    integer x ;
```

```

        constraint con1 {
            myfield == 4;
        }
endextends

```

Examples of advice code

In **Example 15-15**, the extends directive adds advices to the packet::send method.

Example 15-15 :

```

// Begin file example.sv

program test;
    packet p;
    p = new();
    p.send();
endprogram

class packet;
    task send();
        $display("Sending data\n");
    endtask
endclass

extends myaspect(packet);
before task send();
    $display("Before sending packet\n");
endtask

after task send();
    $display("After sending packet\n");
endtask
endextends

// End file example.sv

```

When Example 15-15 is executed, the output is:

```

Before sending packet
Sending data
After sending packet

```

In Example 15-16, extends directive myaspect adds advice to turn off constraint c1 before each call to the foo::pre_randomize method.

Example 15-16 :

```
class foo;
    rand integer myfield;
    constraint c1 {
        myfield == 4;
    }

endclass
extends myaspect(foo);
before task pre_randomize();
    constraint_mode(OFF, "c1")
endtask
endextends
```

In Example 15-15, extends directive myaspect adds advice to set a property named valid to 0 after each call to the foo::post_randomize method.

Example 15-17 :

```
class foo;
    integer valid;
    rand integer myfield;
    constraint c1 {
        myfield == 4;
    }
endclass

extends myaspect(foo);
after task post_randomize();
    valid = 0;
endtask
endextends
```

Example 15-17 shows an aspect extension that defines an around advice for the class method packet::send. When the code in example is compiled and run, the around advice code is executed instead of original packet::send code.

Example 15-18

```
// Begin file example.sv

program test;
    packet p;
    p = new();
    p.setLen(5000);
    p.send();
    p.setLen(10000);
    p.send();
endprogram

class packet;
    integer len;
    task setLen( integer i);
        len = i;
    }
    task send();
        $display("Sending data\n");
    endtask
endclass

extends myaspect(packet);
    around task send();
        if (len < 8000){
            proceed;
        }
        else{
            $display("Dropping packet\n");
        }
    endtask
endextends

// End file example.sv
```

This Example 15-18 also demonstrates how the around advice code can reference properties such as len in the packet object p. When executed the output of the above example is,

```
Sending data
Dropping packet
```

16

Constraints Debugging and Profiling

This chapter explains VCS support for the following constraints features:

- “Using Constraint Profiling” on page 2
- “Using the Constraint Profiling Report” on page 4
- “Using the Hierarchical Constraint Debugger Report” on page 6
- “Array and XMR Support in std::randomize()” on page 8
- “XMR Support in Constraints” on page 11

Using Constraint Profiling

You can use VCS constraint profiling reports to find out how much runtime and memory is spent on each `randomize` call in your testbench. Profiling reports also show cumulative statistics and allow you to cross-probe into the hierarchical constraint debugger reports.

A two-step process is used to profile and trace constraint data:

1. Run a first simulation to generate a profile and a list of the top `randomize` calls. For example:

```
% ./simv +ntb_solver_debug=profile
```

When this first simulation runs, VCS automatically creates a `cstr_html/serial2trace.txt` file which lists the top 10 serial and partition numbers for memory and runtime used. You can optionally add more serial and partition numbers to that file. If you don't want to run this first simulation with constraint profiling on, you can manually create a `serial2trace.txt` file in the `$cwd/cstr_html` directory with the randomize and partition text. To report all randomize serial numbers in the simulation log file, use the following runtime switch:

```
% ./simv +ntb_solver_debug=serial
```

2. Run a second simulation to read this `serial2trace.txt` file and generate detailed XML report data for the listed randomize calls. Use the following switch to generate the XML reports:

```
% ./simv +ntb_solver_debug=trace
```

You can further control the data generated in the XML report using the following switch. Note that the `+ntb_solver_debug=trace` switch must also be on for this next switch to have any effect:

```
./simv +ntb_solver_debug=trace  
+ntb_solver_trace+all+partInfo+unfiltered+filtered+elaborated+partition+result
```

where:

- `all` = report everything
- `partInfo` = report only basic information for each partition
- `unfiltered` = generate an unfiltered trace
- `filtered` = generate a filtered trace
- `elaborated` = generate a GP_problem tag
- `partition` = partitions
- `result` = report randomize_result

Use as many or few of the plus arguments in the switch as you want, depending on what you want to see in the XML reports.

Using the Constraint Profiling Report

After the simulation completes, invoke an HMTL browser on the \$cwd/cstr_html/profile.xml file. For example:

```
% firefox cstr_html/profile.xml
```

[Figure 16-1](#) shows some sample profiler results, including randomize calls that are consuming the most time and memory. Note that this illustration shows just the top of the report.

From the report, you can click the randomize call identifier or the partition identifier to cross-probe to its corresponding constraint hierarchical constraint trace section.

Figure 16-1 VCS Constraint Profiling Example

monarch (us01term6) (win#1) - MetaFrame Presentation Server Client					
Total randomize time: 0.000 seconds					
Total randomize count: 2					
Largest memory increment: 160 KB					
Top randomize calls based on cpu runtime					
File:line@visit	serial#	time	variables	constraints	cnst blocks
test.sv:20@1	1	0.000	2	2	1
test.sv:20@2	2	0.000	2	2	1
Top randomize calls based on cumulative cpu runtime					
File:line	calls	time			
test.sv:20	2	0.000			
Top partitions based on cpu time					
File:line@visit	Rand*GP Serial#	cpu time	variables	constraints	
test.sv:20@1	1*1	0.00	1	1	
test.sv:20@1	1*2	0.00	1	1	

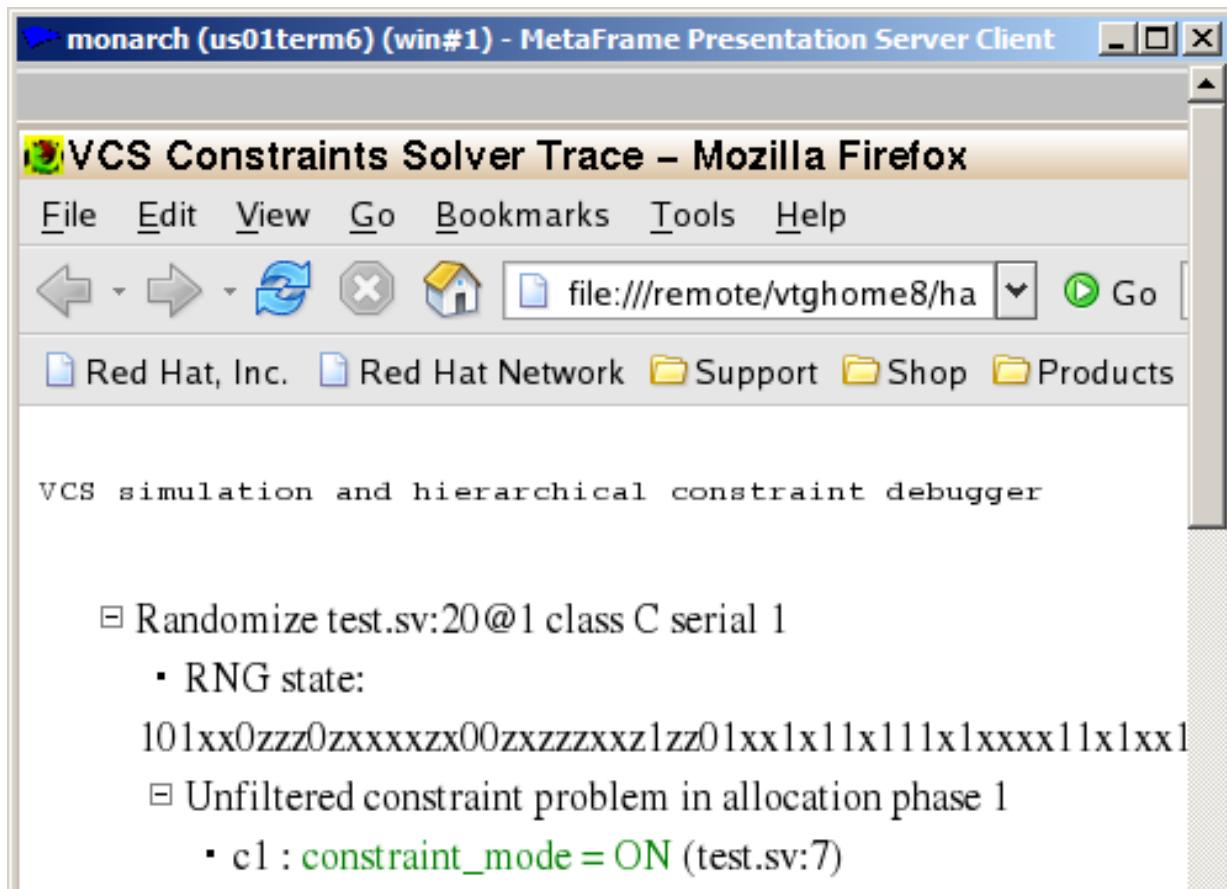
Using the Hierarchical Constraint Debugger Report

After the simulation completes, invoke an HMTL browser on the \$cwd/cstr_html/trace.xml file. For example:

```
% firefox cstr_html/trace.xml
```

[Figure 16-2](#) shows some sample hierarchical trace debugger results. Note that this illustration shows just the top of the report.

Figure 16-2 VCS Hierarchical Constraint Debugger Report



When you first bring up the hierarchical constraint trace, all the items are collapsed. Click a specific item to make it expand. For each `obj.randomize` call, VCS prints the:

- File name and line number in your SystemVerilog or OpenVera source code where that `randomize` call is made.
- Visit count, incremented each time the same `randomize` call occurs. For example, a `randomize` call inside a `for` loop has `test.sv:20@1`, `test.sv:20@2`, and so on. This visit count is also referenced in the profiling tables (see [Figure 16-1](#)).
- File name and line number for each variable to point to the place where it is defined.
- File name and line number of the class where each variable is defined.
- Runtime and memory usage for each partition, and for each full `randomize` call.

Color Coding Constraint Blocks and rand vars

In the randomize reports, each `randomize` call has a number of constraint blocks and variables. VCS uses color coding to identify constraint blocks and rand vars that are turned ON, and different colors for those that are turned OFF.

Avoiding Duplicate Printing of Original Constraint Set

VCS does not print the original set of constraints twice when the constraint debugger or `solver_trace=2` are turned ON. The default printing mode only displays the failure subset instead of both (see [Figure 16-3](#)).

Figure 16-3 Constraint Debugging Report

```
Solver failed when solving following set of constraints
rand integer y; // rand_mode = ON
rand integer z; // rand_mode = ON
rand integer x; // rand_mode = ON
constraint c    // (from this) (constraint_mode = ON)
{
    ( x < 1 ) ;
    ( x in { 3 , 5 , 7 : 11 } ) ;
}
```

You can use the

+ntb_enable_solver_trace_on_failure=0,1,2,3 runtime option as follows:

- 0: Print a one-line failure message with no details.
- 1: Print only the failure subset (this is the default).
- 2: Print the entire constraint problem and failure subset.
- 3: Print only the failure problem. This is useful when the solver fails to determine the minimum subset.

Array and XMR Support in std::randomize()

VCS allows you to use cross-module references (XMRs) in class constraints and inline constraints, in all applicable contexts. Here, XMR means a variable with static storage (anything accessed as a global variable).

VCS std::randomize() support has been enhanced to allow the use of arrays and cross-module references (XMRs) as arguments.

VCS supports all types of arrays:

- fixed-size arrays
- associative arrays
- dynamic arrays
- multidimensional arrays
- smart queues

Note:

VCS does not support multidimensional, variable-sized arrays.

Array elements are also supported as arguments to
`std::randomize()`.

VCS supports all types of XMRs:

- class XMRs
- package XMRs
- interface XMRs
- module XMRs
- static variable XMRs
- any combination of the above

You can use arrays, array elements, and XMRs as arguments to
`std::randomize()`.

Syntax

```
integer fa[3] ;
success= std::randomize(fa) ;
success= std::randomize(fa[2]) ;
```

```
success= std::randomize(pkg::xmr) ;
```

Example

```
module test;
integer i, success;
integer fa[3];
initial
begin
    foreach(fa[i]) $display("%d %d\n", i, fa[i]);
    success = std::randomize(fa);
    foreach(fa[i]) $display("%d %d\n", i, fa[i]);
end
endmodule
```

When `std::randomize()` is called, VCS ignores any rand mode specified on class member arrays or array elements that are used as arguments. This is consistent with how `std::randomize()` is specified in the SystemVerilog LRM. This means that for purposes of `std::randomize()` calls, all arguments have rand mode ON, and none of them are `randc`.

Error Conditions

If you specify an argument to a `std::randomize()` array element which is outside the range of the array, VCS prints the following error message:

```
Error-[CNST-VOAE] Constraint variable outside array error
```

Random variables are not allowed as part of an array index.

If you specify an XMR argument in a `std::randomize()` call, and that XMR that cannot be resolved, VCS prints an error message.

XMR Support in Constraints

You can use XMRs in class constraints and inlined constraints. You can refer to XMR variables directly or by specifying the full hierarchical name, where appropriate. You can use XMRs for all data types, including scalars, enums, arrays, and class objects.

VCS supports all types of XMRs:

- class XMRs
- package XMRs
- interface XMRs
- module XMRs
- static variable XMRs
- any combination of the above

Syntax

```
constraint general
{
    varxmr1 == 3;
    pkg::varxmr2 == 4;
}

c.randomize with { a.b == 5; }
```

Examples

Here is an example of a module XMR:

```
// xmr from module
module mod1;
    int x = 10;
class cls1;
```

```

        rand int i1 [3:0];
        rand int i2;
constraint constr
{
    foreach(i1[a]) i1[a] == mod1.x;
}
endclass

cls1 c1 = new();
initial
begin
    c1.randomize() with {i2 == mod1.x + 5;};
end
endmodule

```

Here is an example of a package XMR:

```

package pkg;
    typedef enum {WEAK,STRONG} STRENGTH;
    class C;
        static rand STRENGTH stren;
    endclass

    pkg::C inst = new;
endpackage

module test;
    import pkg::*;
    initial
    begin
        inst.randomize() with {pkg::C::stren == STRONG;};
        $display("%d", pkg::C::stren);
    end
endmodule

```

Functional Clarifications

XMR resolution in constraints (that is, choosing to which variable VCS binds an XMR variable) is consistent with XMR resolution in procedural SystemVerilog code. VCS first tries to resolve an XMR

reference in the local scope. If the variable is not found in the local scope, VCS searches for it in the immediate upper enclosing scope, and so on, until it finds the variable.

If you specify an XMR variable that cannot be resolved in any parent scopes of the constraint/scope where it is used, VCS errors out and prints an error message.

17

Extensions for SystemVerilog Coverage

The extensions for SystemVerilog coverage include the following:

- “Support for Reference Arguments in `get_coverage()`”
- “Functional Coverage Methodology Using the SystemVerilog C/C++ Interface”

Support for Reference Arguments in `get_coverage()`

The Systemverilog LRM provides several pre-defined methods for every covergroup/coverpoint/cross. See “Predefined Coverage Methods” in Clause 18 of the *SystemVerilog Language Reference Manual for VCS/VCS MX* for information. Two of these pre-defined methods, `get_coverage()` and `get_inst_coverage()`, support optional arguments.

You can use the `get_coverage()` and `get_inst_coverage()` predefined methods to query on coverage during the simulation run, so that you can react to the coverage statistics dynamically.

The `get_coverage()` and `get_inst_coverage()` methods both accept, as optional arguments, a pair of integer values passed by reference.

get_inst_coverage() method

When the optional arguments are entered with the method in coverpoint scope or cross scope, the `get_inst_coverage()` method assigns to the first argument the value of the covered bins, and assigns to the second argument the number of bins for the given coverage item. These two values correspond to the numerator and the denominator used for calculating the coverage score (before scaling by 100).

In covergroup scope, the `get_inst_coverage()` method assigns to the first argument the weighted sum of coverpoint and cross coverage, rounded to the nearest integer, and assigns to the second argument the sum of the weights of the coverpoint or cross items.

get_coverage() method

The numerator and denominator assigned by the `get_coverage()` method depend on the scope.

In covergroup scope, `get_coverage()` assigns to its first argument the weighted sum of the coverage of merged coverpoints and crosses.

In coverpoint or cross scope the first argument to `get_coverage()` is assigned the number of covered bins in the merged coverpoint or cross, and the second argument is assigned the total number of bins.

In all cases, weighted sums are rounded to the nearest integer and the second argument is set to the sum of weights.

Functional Coverage Methodology Using the SystemVerilog C/C++ Interface

This section describes a SystemVerilog-based functional coverage flow. The flow supports functional coverage features—data collection, reporting, merging, grading, analysis, GUI, and so on.

The SystemVerilog functional coverage flow has the following features:

- Performs RTL coverage using covergroups and cover properties.
- Performs C coverage using covergroups.
- Integrates easily with the existing testbench environment.
- Provides coverage analysis capabilities—reporting, grading merging, and GUI.
- Has no negative impact on RTL simulation performance.

Functional coverage is very important in verifying correct functionality of a design. SystemVerilog natively supports functional coverage in RTL code.

However, because C/C++ code is now commonly used in a design (with PLI, DPI, DirectC, and so on), there is no systematic approach to verify the functionality of C/C++.

The SystemVerilog C/C++ interface feature provides an application programming interface (API) so that C/C++ code can use the SystemVerilog functional coverage infrastructure to verify its coverage.

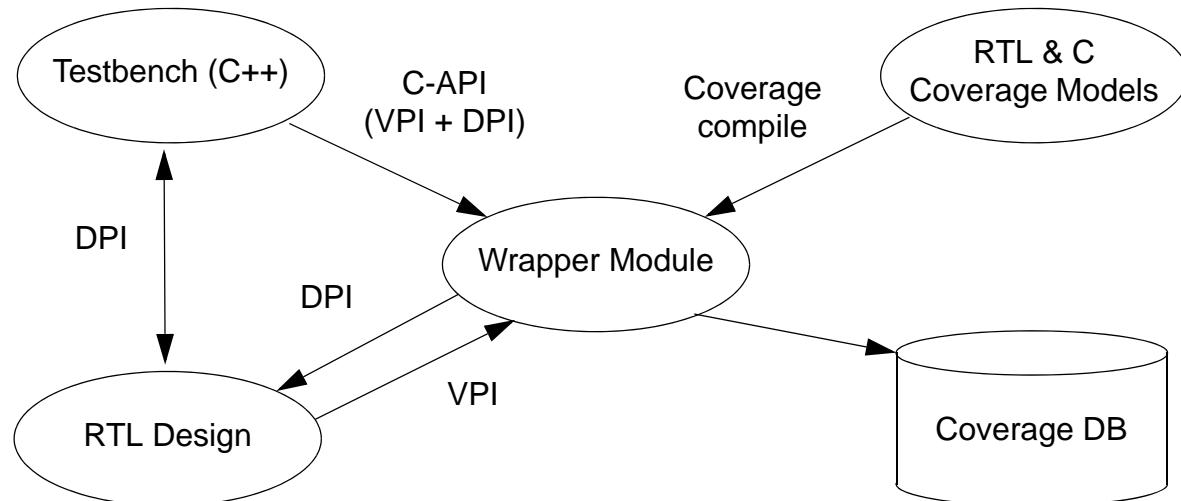
Note:

When you use the SystemVerilog C/C++ interface feature, you need include the header file `svCovgAPI.h`.

SystemVerilog Functional Coverage Flow

[Figure 17-1](#) illustrates the functional coverage flow:

Figure 17-1 SystemVerilog C/C++ Functional Coverage Flow



DPI is the SystemVerilog Direct Programming Interface. See “SystemVerilog DPI” in the *SystemVerilog Language Reference Manual* for VCS/VCS MX for details and examples of using DPI.

VPI is the Verilog Procedural Interface. See “SystemVerilog VPI Object Model” in the *SystemVerilog Language Reference Manual* for VCS/VCS MX for information about using VPI with SystemVerilog.

Covergroups are defined in SystemVerilog, and then they are used to track the functional coverage of C/C++ code through the C-API (C Application Programming Interface). There are two major parts to C/C++ functional coverage interface:

- Covergroup(s)
 - The C/C++ testbench using those covergroups
-

Covergroup Definition

The following section lists the covergroup limitations for C/C++ functional coverage. Covergroups

- Cannot have a sampling clock.
- Must be declared in \$unit.
- Cannot be inside another scope (for example, modules, programs, and so on).
- Must not be instantiated anywhere in else SystemVerilog code.
- Arguments can only be in int, enum (base type int), and bit vector types. The SystemVerilog-to-C data-type mapping is compliant with DPI. [Table 17-1](#) shows the mapping of the supported types:

Table 17-1 SystemVerilog-to-C Data-Type Mapping by DPI

SystemVerilog	C
int	int
bit	unsigned char
bit [m:n]	svBitVec32
enum int	int

- Definitions must appear in files that are separate from the DUT because the definitions are compiled separately with the VCS command-line option `-c_covg`.

After you define the covergroups, compile them with `-c_covg` (that is, `-c_covg <covergroup_file>`). If you have multiple covergroup files, you must precede each of them with the `-c_covg` option (that is, `-c_covg <cov_file1> -c_covg <cov_file2> ...`).

The options `-sverilog`, `-ntb_opts dtm`, and `+vpi` are also needed when compiling with `-c_covg`.

After compiling the covergroups to be used with C/C++, the C-API allows for the allocation of covergroup handles, manual triggering of the covergroup sample, and the ability to de-instance and free the previously declared covergroup handle.

The following is a list of the C-API functions:

- `svCovgNew / svCovgNew2`
- `svCovgSample / svCovgSample2`
- `svCovgDelete`

Detailed specifications for these functions appear in “[C/C++ Functional Coverage API Specification](#)” .

The following examples demonstrate the use model.

SystemVerilog (Covergroup for C/C++): covg.sv

```
cp: coverpoint count {
    bins b = {data};
    ...
}
endgroup
```

C Testbench: test.c

```
int my_c_testbench ()
{
    svCovgHandle cgh;
    // C variables
    int data;
    int count;
```

Approach #1: Passing Arguments by Reference

```
// Create a covergroup instance; pass data as a value
// parameter and count as a reference parameter;
// coverage handle remembers references
cgh = svCovgNew("cg", "cg_inst", SV_SAMPLE_REF, data,
&count);

// Sample stored references
svCovgSample(cgh); // sampling by the stored reference
...

// Delete covergroup instance
svCovgDelete(cgh);
```

Approach #2: Passing Arguments by Value

```
// Create a covergroup instance; pass data and count as
// value parameters
```

```

cgh = svCovgNew("cg", "cg_inst", SV_SAMPLE_VAL, data,
count);

// Sample values passed for covergroup ref arguments
svCovgSample(cgh, count); // sampling the value of count
...
// Delete covergroup instance
svCovgDelete(cgh);

```

Compile Flow

Compile the coverage model (`covg.sv`) using `-c_covg` together with the design and the C testbench

This step assumes that you invoke the C testbench from the design `dut.sv` through some C interface (for example, DPI, PLI, and so on). For example:

```
vcs -sverilog dut.sv test.c -c_covg -ntb_opts dtm +vpi covg.sv
```

Runtime

At runtime (executing `simv`), the functional coverage data is collected and stored in the coverage database.

C/C++ Functional Coverage API Specification

This section gives detailed specifications for the C/C++ functional coverage C-API.

```
svCovgHandle svCovgNew (char* cgName, char* ciName, int refType, args ...);
```

```
svCovgHandle svCovgNew2 (char* cgName, char* ciName, int refType, va_list vl);
```

Parameters

cgName

Covergroup name.

ciName

Covergroup instance name (should be unique).

refType

SV_SAMPLE_REF or SV_SAMPLE_VAL.

args...

A variable number of arguments for creating a new covergroup instance.

vl

Represents a C predefined data structure (va_list) for maintaining a list of arguments.

Description

Create a covergroup instance using the covergroup and instance names. If no error, return `svCovgHandle`, otherwise return NULL. The C variable sampling type (either reference or value) is specified using `refType`. The sampling type is stored in `svCovgHandle`. The

`svCovgNew2` function is similar to `svCovgNew` except that you provide it with a `va_list`, instead of a variable number of arguments (represented by "...") to `svCovgNew`.

For value sampling, pass values for non-reference and reference arguments in the order specified in the covergroup declaration, and set `refType` to `SV_SAMPLE_VAL`.

For reference sampling, pass values for non-reference arguments and addresses for reference arguments in the order specified in the covergroup declaration. References must remain valid during the life of the covergroup instance. Set `refType` to `SV_SAMPLE_REF`.

Type checking is not performed for arguments. It is your responsibility to pass correct values and addresses.

int svCovgSample(svCovgHandle *ch*, args ...);

int svCovgSample2(svCovgHandle *ch*, va_list *vl*);

Parameters

ch

Handle to a covergroup instance created by `svCovgNew()`.

args ...

A variable number of arguments for sampling a covergroup by value, if `refType = SV_SAMPLE_VAL` in `svCovgNew()`.

vl

Represents a C predefined data structure (`va_list`) for maintaining a list of arguments.

Description

Sample a covergroup instance using the sampling style stored in `svCovgHandle` and return 1 (TRUE) if no error, otherwise return 0 (FALSE). The `svCovgSample2` function is similar to `svCovgSample` except that you provide a `va_list`, instead of a variable number of arguments (represented by "..."), to `svCovgSample`.

For value sampling, provide values for reference arguments in the order specified in the covergroup declaration. Type checking is not performed for value arguments. It is your responsibility to pass correct values.

For reference sampling, use stored addresses for reference arguments in `svCovgHandle`.

int svCovgDelete(svCovgHandle *ch*);

Parameters

ch

Handle to a covergroup instance created by `svCovgNew()` (or `svCovgNew2`).

Description

Delete a covergroup instance and return 1 (TRUE) if no error, otherwise return 0 (FALSE).

18

OpenVera-SystemVerilog Testbench Interoperability

The primary purpose of OpenVera-SystemVerilog interoperability in VCS Native Testbench is to enable you to reuse OpenVera classes in new SystemVerilog code without rewriting OpenVera code into SystemVerilog.

This chapter describes:

- “[Scope of Interoperability](#)”
- “[Importing OpenVera types into SystemVerilog](#)”

Using the SystemVerilog package import syntax to import OpenVera data types and constructs into SystemVerilog.

- “Data Type Mapping”

The automatic mapping of data types between the two languages as well as the limitations of this mapping (some data types cannot be directly mapped).

- “Connecting to the Design”

Mapping of SystemVerilog modports to OpenVera where they can be used as OpenVera virtual ports.

- “Notes to Remember”

- “Usage Model”

- “Limitations”

Scope of Interoperability

The scope of OpenVera-SystemVerilog interoperability in VCS Native Testbench is as follows:

- Classes defined in OpenVera can be used directly or extended in SystemVerilog testbenches.
- Program blocks must be coded in SystemVerilog. The SystemVerilog interface can include constructs like modports and clocking blocks to communicate with the design.
- OpenVera code must not contain program blocks, bind statements, or predefined methods. It can contain classes, enums, ports, interfaces, tasks, and functions.

- OpenVera code can use virtual ports for sampling, driving, or waiting on design signals that are connected to the SystemVerilog testbench.

Importing OpenVera types into SystemVerilog

OpenVera has two user-defined types: enums and classes. These types can be imported into SystemVerilog by using the SystemVerilog package import syntax:

```
import OpenVera::openvera_class_name;  
import OpenVera::openvera_enum_name;
```

Allows one to use `openvera_class_name` in SystemVerilog code in the same way as a SystemVerilog class. This includes the ability to:

- Create objects of type `openvera_class_name`
- Access or use properties and types defined in `openvera_class_name` or its base classes
- Invoke methods (virtual and non-virtual) defined in `openvera_class_name` or its base classes
- Extend `openvera_class_name` to SV classes

However, this does not import the names of base classes of `openvera_class_name` into SystemVerilog (that requires an explicit import). For example:

```
// OpenVera
    class Base {
        .
        .
        .
        task foo(arguments) {
            .
            .
            .
        }
        virtual task (arguments) {
            .
            .
            .
        }
    class Derived extends Base {
        virtual task vfoo(arguments) {
            .
            .
            .
        }
    }

// SystemVerilog
import OpenVera::Derived;
Derived d = new; // OK
initial begin
    d.foo();      // OK (Base::foo automatically
                  // imported)
    d.vfoo();     // OK
end
Base b = new; // not OK (don't know that Base is a
              // class name)
```

The previous example would be valid if you add the following line before the first usage of the name Base.

```
import OpenVera::Base;
```

Continuing with the previous example, SystemVerilog code can extend an OpenVera class as shown below:

```
// SystemVerilog
import OpenVera::Base;
class SVDerived extends Base;
    virtual task vmt()
        begin
            .
            .
            .
        end
    endtask
endclass
```

Note:

- If a derived class redefines a base class method, the arguments of the derived class method must exactly match the arguments of the base class method.

- Explicit import of each data type from OpenVera can be avoided by a single `import OpenVera::*;`

```
// OpenVera
    class Base {
        integer i;
        .
        .
        .
    }
    class wrappedBase {
        public Base myBase;
    }
// SystemVerilog
import OpenVera::wrappedBase;
class extendedWrappedBase extends wrappedBase;
.
.
.
endclass
```

In this example, `myBase.i` can be used to refer to this member of `Base` from the SV side. However, if SV also needs to use objects of type `Base`, then you must include:

```
import OpenVera::Base;
```

Data Type Mapping

This section describes how various data types in SystemVerilog are mapped to OpenVera and vice-versa:

- *Direct mapping:* Many data types have a direct mapping in the other language and no conversion of data representation is required. In such cases, we say that the OpenVera type is equivalent to the SystemVerilog type.

- *Implicit conversion:* In other cases, VCS performs implicit type conversion. The rules of inter-language implicit type conversion follows the implicit type conversion rules specified in SystemVerilog LRM. To apply SystemVerilog rules to OpenVera, the OpenVera type must be first mapped to its equivalent SystemVerilog type. For example, there is no direct mapping between OpenVera `reg` and SystemVerilog `bit`. But `reg` in OpenVera can be directly mapped to `logic` in SystemVerilog. Then the same implicit conversion rules between SystemVerilog `logic` and SystemVerilog `bit` can be applied to OpenVera `reg` and SystemVerilog `bit`.
- *Explicit translation:* In the case of mailboxes and semaphores, the translation must be explicitly performed by the user. This is because in OpenVera, mailboxes and semaphores are represented by `integer` *ids* and VCS cannot reliably determine if an `integer` value represents a mailbox *id*.

Mailboxes and Semaphores

Mailboxes and semaphores are referenced using object handles in SystemVerilog whereas in OpenVera they are referenced using integral *ids*.

VCS supports the mapping of mailboxes between the two languages.

For example, consider a mailbox created in SystemVerilog. To use it in OpenVera, you need to get the *id* for the mailbox somehow. The `get_id()` function, available as a VCS extension to SV, returns this value:

```
function int mailbox::get_id();
```

It will be used as follows:

```
// SystemVerilog
    mailbox mbox = new;
    int id;
    .
    .
    .
    id = mbox.get_id();
    .
    .
    .
    .
    .
    foo.vera_method(id);

// OpenVera
class Foo {
    .
    .
    .
    task vera_method(integer id) {
        .
        .
        .
        void = mailbox_put(data_type mailbox_id,
                            data_type variable);
    }
}
```

Once OpenVera gets an *id* for a mailbox/semaphore it can save it into any `integer` type variable. Note that however if `get_id` is invoked for a mailbox, the mailbox can no longer be garbage collected because VCS has no way of knowing when the mailbox ceases to be in use.

Typed mailboxes (currently not supported), when they are supported in SystemVerilog can be passed to OpenVera code using the same method as untyped mailboxes above. However, if the OpenVera code attempts to put an object of incompatible type into a typed mailbox, a simulation error will result.

Bounded mailboxes (currently not supported), when they are supported in SystemVerilog can be passed to OpenVera code using the same method as above. OpenVera code trying to do `mailbox_put` into a full mailbox will result in a simulation error.

To use an OpenVera mailbox in SystemVerilog, you need to get a handle to the mailbox object using a system function call. The system function `$get_mailbox` returns this handle:

```
function mailbox $get_mailbox(int id);
```

It will be used as follows:

```
// SystemVerilog
.
.
.
mailbox mbox;
int id = foo.vera_method();      // vera_method returns an
                                // OpenVera mailbox id
mbox = $get_mailbox(id);
```

Analogous extensions are available for semaphores:

```
function int semaphore::get_id();
function semaphore $get_semaphore(int id);
```

Events

The OpenVera event data type is equivalent to the SystemVerilog event data type. Events from either language can be passed (as method arguments or return values) to the other language without any conversion. The operations performed on events in a given language are determined by the language syntax:

An event variable can be used in OpenVera in `sync` and `trigger`. An event variable `event1` can be used in SystemVerilog as follows:

```
event1.triggered //event1 triggered state property  
->event1      //trigger event1  
@(event1)      //wait for event1
```

Strings

OpenVera and SystemVerilog strings are equivalent. Strings from either language can be passed (as method arguments or return values) to the other language without any conversion. In OpenVera, `null` is the default value for a `string`. In SystemVerilog, the default value is the empty string (""). It is illegal to assign `null` to a `string` in SystemVerilog. Currently, NTB-OV treats "" and `null` as distinct constants (equality fails).

Enumerated Types

SystemVerilog enumerated types have arbitrary base types and are not generally compatible with OpenVera enumerated types. A SystemVerilog enumerated type will be implicitly converted to the

base type of the enum (an integral type) and then the bit-vector conversion rules (section 2.5) are applied to convert to an OpenVera type. This is illustrated in the following example:

```
// SystemVerilog
    typedef reg [7:0] formal_t; // SV type equivalent to
                                // 'reg [7:0]' in OV
    typedef enum reg [7:0] { red = 8'hff, blue = 8'hfe,
                            green = 8'hfd } color;
    // Note: the base type of color is 'reg [7:0]'
    typedef enum bit [1:0] { high = 2'b11, med = 2'b01,
                            low = 2'b00 } level;
    color c;
    level d = high;
    Foo foo;
    ...
    foo.vera_method(c); // OK: formal_t'(c) is passed to
                        // vera_method.
    foo.vera_method(d); // OK: formal_t'(d) is passed to
                        // vera_method.
                        // If d == high, then 8'b00000011 is
                        // passed to vera_method.

// OpenVera
    class Foo {
        ...
        task vera_method(reg [7:0] r) {
            ...
        }
    }
}
```

The above data type conversion does not involve a conversion in data representation. An enum can be passed by reference to OpenVera code but the formal argument of the OpenVera method must exactly match the enum base type (for example: 2-to-4 value conversion, sign conversion, padding or truncation are not allowed for arguments passed by reference; they are OK for arguments passed by value).

Enumerated types with 2-value base types will be implicitly converted to the appropriate 4-state type (of the same bit length). See the discussion in 2.5 on the conversion of bit vector types.

OpenVera enum types can be imported to SystemVerilog using the following syntax:

```
import OpenVera::openvera_enum_name;
```

It will be used as follows:

```
// OpenVera
    enum OpCode { Add, Sub, Mul } ;

// System Verilog
    import OpenVera::OpCode;
    OpCode x = OpenVera::Add;

// or the enum label can be imported and then used
// without OpenVera::

    import OpenVera::Add;
    OpCode y = Add;
```

Note: SystemVerilog enum methods such as next, prev and name can be used on imported OpenVera enums.

Enums contained within OV classes are illustrated in the following example:

```
class OVclass{
    enum Opcode {Add, Sub, Mul} ;
}

import OpenVera::OVclass;
OVclass::Opcode SVvar;
SVvar=OVclass::Add;
```

Integers and Bit-Vectors

The mapping between SystemVerilog and OpenVera integral types are shown in the following table:

SystemVerilog	OpenVera	2/4 or 4/2 value conversion?	Change in sign?
integer	integer	N (equivalent types)	N (Both signed)
byte	reg [7:0]	Y	Y
shortint	reg [15:0]	Y	Y
int	integer	Y	N (Both signed)
longint	reg [63:0]	Y	Y
logic [m:n]	reg [abs(m-n)+1:0]	N (equivalent types)	N (Both unsigned)
bit [m:n]	reg [abs(m-n)+1:0]	Y	N (Both unsigned)
time	reg [63:0]	Y	N (Both unsigned)

Note:

If a value or sign conversion is needed between the actual and formal arguments of a task or function, then the argument cannot be passed by reference.

Arrays

Arrays can be passed as arguments to tasks and functions from SystemVerilog to OpenVera and vice-versa. The formal and actual array arguments must have equivalent element types, the same number of dimensions with corresponding dimensions of the same length. These rules follow the SystemVerilog LRM.

- A SystemVerilog fixed array dimension of the form [m:n] is directly mapped to [abs(m-n)+1] in OpenVera.

- An OpenVera fixed array dimension of the form `[m]` is directly mapped to `[m]` in SystemVerilog.

Rules for equivalency of other (non-fixed) types of arrays are as follows:

- A dynamic array (or Smart queue) in OpenVera is directly mapped to a SystemVerilog dynamic array if their element types are equivalent (can be directly mapped).
- An OpenVera associative array with unspecified key type (for example `integer a []`) is equivalent to a SystemVerilog associative array with key type `reg [63:0]` provided the element types are equivalent.
- An OpenVera associative array with `string` key type is equivalent to a SystemVerilog associative array with `string` key type provided the element types are equivalent.

Other types of SystemVerilog associative arrays have no equivalent in OpenVera and hence they cannot be passed across the language boundary.

Some examples of compatibility are described in the following table:

OpenVera	SystemVerilog	Compatibility
<code>integer a[10]</code>	<code>integer b[11:2]</code>	Yes
<code>integer a[10]</code>	<code>int b[11:2]</code>	No
<code>reg [11:0] a[5]</code>	<code>logic [3:0] [2:0] b[5]</code>	Yes

A 2-valued array type in SystemVerilog cannot be directly mapped to a 4-valued array in OpenVera. However, a cast may be performed as follows:

```
// OpenVera
    class Foo {
        .
        .
        .
        task vera_method(integer array[5]) {
            .
            .
            .
            .
            .
            .
        }
// SystemVerilog
    int array[5];
    typedef integer array_t[5];
    import OpenVera::Foo;
    Foo f;
    .
    .
    .
    f.vera_method(array);           // Error: type mismatch
    f.vera_method((array_t')array)); // OK
    .
    .
    .
```

Structs and Unions

Unpacked structs/unions cannot be passed as arguments to OpenVera methods. Packed structs/unions can be passed as arguments to OpenVera: they will be implicitly converted to bit vectors of the same width.

```
packed struct { ... } s in SystemVerilog is mapped to  
reg [m:0] r in OpenVera where m == $bits(s).
```

Analogous mapping applies to unions.

Connecting to the Design

Mapping Modports to Virtual Ports

This section relies on the following extensions to SystemVerilog supported in VCS.

Virtual Modports

VCS supports a reference to a modport in an interface to be declared using the following syntax.

```
virtual interface_name.modport_name virtual_modport_name;
```

For example:

```
interface IFC;  
    wire a, b;  
    modport mp (input a, output b);  
endinterface  
  
IFC i();  
virtual IFC.mp vmp;  
. . .  
vmp = i.mp;
```

Importing Clocking Block Members into a Modport

VCS allows a reference to a clocking block member to be made by omitting the clocking block name.

For example, in SystemVerilog a clocking block is used in a modport as follows:

```
interface IFC(input clk);
    wire a, b;
    clocking cb @(posedge clk);
        input a;
        input b;
    endclocking
    modport mp (clocking cb);
endinterface

program mpg(IFC ifc);
    .
    .
    .
    virtual IFC.mp vmp;
    .
    .
    .
    vmp = i.mp;
    @(vmp.cb.a); // here we need to specify cb explicitly
    .
endprogram
module top();
    .
    .
    .
    IFC ifc(clk); // use this to connect to DUT and TB
    mpg mpg(ifc);
    dut dut(...);
    .
endmodule
```

VCS supports the following extensions that allow the clocking block name to be omitted from vmp.cb.a.

```
// Example-1
interface IFC(input clk);
    wire a, b;
    clocking cb @(posedge clk);
        input a;
        input b;
    endclocking
    modport mp (import cb.a, import cb.b);
endinterface

program mpg(IFC ifc);
.
.
.
virtual IFC.mp vmp;
.
.
.
vmp = i.mp;
@(vmp.a); // cb can be omitted; 'cb.a' is
           // imported into the modport

.
endprogram
module top();
.
.
.
IFC ifc(clk); // use this to connect to DUT and TB
mpg mpg(ifc);
dut dut(...);
.
.
.
endmodule

// Example-2
interface IFC(input clk);
    wire a, b;
    bit clk;
    clocking cb @(posedge clk);
```

```

        input a;
        input b;
    endclocking
    modport mp (import cb.*); // All members of cb
                                // are imported.
                                // Equivalent to the
                                // modport in
                                // Example-1.
endinterface

program mpg(IFC ifc);
.
.
.
IFC i(clk);
.
.
.
virtual IFC.mp vmp;
.
.
.
vmp = i.mp;
@(vmp.a); // cb can be omitted;
           //'cb.a' is imported into the modport
endprogram

module top();
.
.
.
IFC ifc(clk); // use this to connect to DUT and TB
mpg mpg(ifc);
dut dut(...);
.
.
.
endmodule

```

A SystemVerilog modport can be implicitly converted to an OpenVera virtual port provided the following conditions are satisfied:

- The modport and the virtual port have the same number of members.
- Each member of the modport converted to a virtual port must either be: (1) a clocking block, or (2) imported from a clocking block using the `import` syntax above.
- For different modports to be implicitly converted to the same virtual port, the corresponding members of the modports (in the order in which they appear in the modport declaration) be of bit lengths. If the members of a clocking block are imported into the modport using the `cb.*` syntax, where `cb` is a clocking block, then the order of those members in the modport is determined by their declaration order in `cb`.

Example

```
// OpenVera
port P {
    clk;
    a;
    b;
}

class Foo {
    P p;
    task new(P p_) {
        p = p_;
    }

    task foo() {
        .
        .
        .
        @(p.$clk);
        .
        variable = p.$b;
        p.$a = variable;
        .
        .
        .
    }
}

// SystemVerilog
interface IFC(input clk);
    wire a;
    wire b;

    clocking clk_cb @(clk);
        input #0 clk;
    endclocking

    clocking cb @(posedge clk);
        output a;
        input b;
    endclocking
}
```

```

modport mp (import clk_cb.* , import cb.*); // modport
    // can aggregate signals from multiple clocking blocks.

endinterface: IFC

program mpg(IFC ifc);
    import OpenVera::Foo;
    .

    .
    virtual IFC.mp vmp = ifc.mp;
    Foo f = new(vmp); // clocking event of ifc.cb mapped to
                      // $clk in port P
                      // ifc.cb.a mapped to $a in port P
                      // ifc.cb.b mapped to $b in port P
    .
    f.foo();
    .
    .
    .
endprogram

module top();
    .
    .
    IFC ifc(clk); // use this to connect to DUT and TB
    mpg mpg(ifc);
    dut dut(...);
    .
    .
endmodule

```

Note:

In the above example, you can also directly pass the `vmp` modport from an interface instance:

```
Foo f = new(ifc.mp);
```

Semantic Issues with Samples, Drives, and Expects

When OpenVera code wants to sample a DUT signal through a virtual port (or interface), if the current time is not at the relevant clock edge, the current thread is suspended until that clock edge occurs and then the value is sampled. NTB-OV implements this behavior by default. On the other hand, in SystemVerilog, sampling never blocks and the value that was sampled at the most recent edge of the clock is used. Analogous differences exist for drives and expects.

Notes to Remember

Blocking Functions in OpenVera

When a SystemVerilog function calls a virtual function that may resolve to a blocking OpenVera function at runtime, the compiler cannot determine with certainty if the SystemVerilog function will block. VCS issues a warning at compile time and let the SystemVerilog function block at runtime.

Besides killing descendant processes in the same language domain, `terminate` invoked from OpenVera will also kill descendant processes in SystemVerilog. Similarly, `disable fork` invoked from SystemVerilog will also kill descendant processes in OpenVera. `wait_child` will also wait for SystemVerilog descendant processes and `wait fork` will also wait for OpenVera descendant processes.

Constraints and Randomization

- SystemVerilog code can call `randomize()` on objects of an OpenVera class type.
- In SystemVerilog code, SystemVerilog syntax must be used to turn off/on constraint blocks or randomization of specific `rand` variables (even for OpenVera classes).
- Random stability will be maintained across the language domain.

```
//OV
class OVclass{
    rand integer ri;
    constraint cnst{...}
}

//SV
OVclass obj=new();
SVclass Sobj=new();
Sobj.randomize();
obj.randomize() with
{obj.ri==Sobj.var;};
```

Functional Coverage

There are some differences in functional coverage semantics between OpenVera and SystemVerilog. These differences are currently being eliminated by changing OpenVera semantics to conform to SystemVerilog. In interoperability mode, `coverage_group` in OpenVera and `covergroup` in SystemVerilog will have the same (SystemVerilog) semantics. Non-embedded coverage group can be imported from Vera to SystemVerilog using the package `import` syntax (similar to classes).

Coverage reports will be unified and keywords such as `coverpoint`, `bins` will be used from SystemVerilog instead of OpenVera keywords.

Here is an example of usage of coverage groups across the language boundary:

```
// OpenVera
class A
{
    B b;
    coverage_group cg {
        sample x(b.c);
        sample y(b.d);
        cross cc1(x, y);
        sample_event = @(posedge CLOCK);
    }
    task new() {
        b = new;
    }
}
// SystemVerilog

import OpenVera::A;

initial begin
    A obj = new;
    obj.cg.option.at_least = 2;
    obj.cg.option.comment = "this should work";
    @(posedge CLOCK);
    $display("coverage=%f", obj.cg.get_coverage());
end
```

Usage Model

Any `define from the OV code will be visible in SV once they are explicitly included.

Note:

OV #define must be rewritten as `define for ease of migration to SV.

Compilation

```
% vcs [compile_options] -sverilog -ntb_opts interop  
[other_NTB_options] file4.sv file5.vr file2.v file1.v
```

Simulation

```
% simv [simv_options]
```

Note:

- If RVM class libs are used in the OV code, use `-ntb_opts rvm` with `vlogan` command line.
- Using `-ntb_opts interop -ntb_opts rvm` with `vcs`, automatically translates `rvm_` macros in OV package to `vmm_` equivalents.

Limitations

Classes extended/defined in SystemVerilog cannot be instantiated by OpenVera. OpenVera verification IP will need to be compiled with the NTB syntax and semantic restrictions. These restrictions are detailed in the *Native Testbench Coding Guide*, included in the VCS release.

SystemVerilog contains several data types that are not supported in OpenVera including real, unpacked-structures, and unpacked-unions. OpenVera cannot access any variables or class data members of these types. A compiler error will occur if the OpenVera code attempts to access the undefined SystemVerilog data member. This does not prevent SystemVerilog passing an object to OpenVera, and then receiving it back again, with the unsupported data items unchanged.

19

Integrating VCS with Debussy

This chapter describes the required environmental settings and the usage model to dump an fsdb file:

- “[Setting Up Debussy](#)”
- “[Usage Model to Dump fsdb File](#)”

Setting Up Debussy

To dump an fsdb file, you need to set the following environment variables:

```
% setenv DEBUSSY_HOME Debussy_installation
% setenv LM_LICENSE_FILE [Debussy_license] :$LM_LICENSE_FILE
```

Usage Model to Dump fsdb File

This section describes the usage model to dump an fsdb file using Verilog system tasks, or UCLI.

- Using Verilog System Tasks

You can use the Verilog system tasks `$fsdbDumpfile()` and `$fsdbDumpvars()` in your Verilog design to dump an fsdb file (see “[Using Verilog System Tasks](#)”).

- UCLI

At UCLI prompt, you can use the UCLI commands `fsdbDumpfile` and `fsdbDumpvars` to dump an fsdb file (see “[Using UCLI](#)”).

Irrespective of whether you are using system tasks or UCLI commands, you must use the compilation option `-fsdb` to enable fsdb dumping as shown below:

Using Verilog System Tasks

Compilation

```
% vcs -fsdb [compile_options] verilog_filelist
```

Simulation

```
% simv [run_options]
```

Using UCLI

Compilation

```
% vcs -fsdb -debug_pp [compile_options] verilog_filelist
```

Simulation

```
% simv [run_options] -ucli  
ucli> fsdbDumpfile your_fsdb_dumpfile  
ucli> fsdbDumpvars level module/entity
```

Example 19-1 Using Verilog System Tasks

This example demonstrates the use of Verilog system tasks, \$fsdbDumpfile and \$fsdbDumpvars.

```
`timescale 1ns\1ns  
module test;  
initial  
begin  
  $fsdbDumpfile("test.fsdb");  
  $fsdbDumpvars(0,test);  
end  
  
...  
endmodule
```

Now the usage model to compile and simulate the above design is as shown below:

Compilation

```
% vcs -fsdb test.v
```

Simulation

```
% simv
```

The above set of commands dumps all the instances in test into the test.fsdb file.

Example 19-2 Using UCLI

This example demonstrates the use of UCLI commands fsdbDumpfile and fsdbDumpvars at the UCLI prompt to dump an fsdb file:

Consider the following Verilog file:

```
'timescale 1ns/1ns
module test();
.....
endmodule
```

The usage model to compile the design to use UCLI commands is as shown below:

Compilation

```
% vcs -fsdb -debug_pp test.v
```

Simulation

```
% simv -ucli
ucli> fsdbDumpfile test.fsdb
ucli> fsdbDumpvars 0 test
ucli> run
ucli> quit
```

The above command dumps the whole design `test` into the `test.fsdb` file.

You can also write the above UCLI commands in a file and use the `-do` runtime option to execute the UCLI commands (see [Chapter 9, "Using Unified Command-line Interface"](#)).

20

Using SystemVerilog Assertions

Using SystemVerilog assertion (SVA) you can specify how you expect a design to behave and have VCS display messages when the design does not behave as specified.

```
assert property (@(posedge clk) req |-> ##2 ack)
               else $display ("ACK failed to follow the request);
```

The above example displays, "ACK failed to follow the request", if ACK is not high two clock cycles after req is high. This example is a very simple assertion. For more information on how to write assertions, see Chapter 17 of *SystemVerilog Language Reference Manual*.

VCS allows you to:

- Control the SVAs
- Enable or Disable SVAs

- Control the simulation based on the assertion results

This chapter describes the following:

- “[Using SVAs in the HDL Design](#)”
- “[Controlling SystemVerilog Assertions](#)”
- “[Viewing Results](#)”

Note:

Synopsys recommends you to use the gcc compiler for Solaris platform.

Using SVAs in the HDL Design

You can instantiate SVAs in your HDL design in the following ways:

- “[Using Standard Checker Library](#)”
 - “[Binding SVA to a Design](#)”
 - “[Inlining SVAs in the Verilog Design](#)”
-

Using Standard Checker Library

VCS provides you SVA checkers, which can be directly instantiated in your Verilog source files. You can find these SVA checkers files in `$_VCS_HOME/packages/sva` directory.

This section describes the usage model to compile and simulate the design with SVA checkers. For more information on SVA checker libraries and list of available checkers, see the *SystemVerilog Assertions Checker Library Reference Manual*.

Instantiating SVA Checkers in Verilog

You can instantiate SVA checkers in your Verilog source just like instantiating any other Verilog module. For example, to instantiate the checker `assert_always`, specify:

```
module my_verilog();
    ...
    always_inst: assert_always(.clk(clk), .reset(rst), a(1));
    ...
endmodule
```

The usage model to simulate the design with SVA checkers is as follows:

Compilation

```
% vcs [vcs_options] -sverilog +define+ASSERT_ON \
+incdir+$VCS_HOME/packages/sva \
Verilog_source_files
```

Simulation

```
% simv [simv_options]
```

For more information on SVA checker libraries and a list of available checkers, see the *SystemVerilog Assertions Checker Library Reference Manual*.

Binding SVA to a Design

Using bind statements to bind SVAs to your Verilog design is another way to use SVAs. The advantage is, bind statements allow you to bind SVAs to Verilog designs without modifying or editing your design files.

The bind statement syntax is as follows:

```
bind inst_name/module SVA_module
      # [SVA_parameters] SVA_inst_name [SVA_ports]
```

The bind statement for Verilog targets can be used anywhere within your Verilog source file. For example:

```
//Verilog file
module dev (...);
...
endmodule

bind dev dev_checker dc1 (.clk(clk), .a(a), .b(b));
```

As shown in the above example, the bind statement is specified in the same Verilog file.

The usage model to simulate the design is as shown below:

Compilation

```
% vcs -sverilog [compile_options] Verilog_files
```

Simulation

```
% simv [run_options]
```

Inlining SVAs in the Verilog Design

For Verilog designs, you can write SVAs as part of the code or within pragmas as shown in the following example:

Example 1: Writing Assertions as a part of the code

```
module dut(...);

    ...

sequence s1;
@(posedge clk) sig1 ##[1:3] sig2;
endsequence

...

endmodule
```

Example 2: Writing Assertions using SVA pragmas (/sv_pragma)

```
module dut(...);

    ...

//sv_pragma sequence s1;
//sv_pragma     @(posedge clk) sig1 ##[1:3] sig2;
//sv_pragma endsequence

/*sv_pragma
sequence s2;
    @(posedge clk) sig3 ##[1:3] sig4;
endsequence
*/
...

endmodule
```

As shown in Example 2, you can use SVA pragmas as //sv pragma at the beginning of all SVA lines, or you can use the following to mark a block of code as SVA code:

```
/* sv pragma
sequence s2;
  @(posedge clk) sig3 ##[1:3] sig4;
endsequence
*/
```

Usage Model

The usage model to compile and simulate the designs having inlined assertions is as follows:

Note:

If you have your assertions inlined using //sv pragma, use the analysis option -sv pragma as shown above.

Compilation

```
% vcs -sva_pragma [compile_options] Verilog_files
```

Simulation

```
% simv [run_options]
```

Controlling SystemVerilog Assertions

SVAs can be controlled or monitored using:

- “Compilation and Runtime Options”
- “Assertion Monitoring System Tasks”
- “Using Assertion Categories”

Compilation and Runtime Options

VCS provides various compilation options to perform the following tasks:

- Enable controlling of assertions during runtime

Use the `-assert enable_diag` option, if you also want to control assertions during runtime. The runtime options are enabled only if you compile the design with this option.

- Dump assertion information in the VPD file and view the assertion information in DVE

You can use the `-assert dve` option to enable dumping assertion information in the VPD file. This option also allows you to view assertion information in the assertion pane in DVE (for additional information, see the *DVE User Guide*.)

- Disable all or a few assertions in the design

You can use the `-assert disable` compilation option to disable all the SVAs in the design, or

`-assert disable_file=file_name` to disable the SVAs specified in the file.

- Disable assertion coverage

By default, when you use the `-cm assert` option during compilation and simulation, VCS enables monitoring your assertions for coverage, and writes an assertion coverage database during simulation. Now, using the option `-assert disable_cover` you can disable assertion coverage.

- Disable dumping of SVA information in the VPD file

You can use the `-assert dumpoff` option to disable the dumping of SVA information to the VPD file during simulation (for additional information, see “[Options for SystemVerilog Assertions](#)” on page B-9).

Following are the tasks VCS allows you to do during the runtime:

- Terminate simulation after certain number of assertion failures

You can use either the `-assert finish_maxfail=N` or `-assert global_finish_maxfail=N` runtime option to terminate the simulation if the number of failures for any assertion reaches N or if the total number of failures from all SVAs reaches N , respectively.

- Show both passing and failing assertions

By default, VCS reports only failures. However, you can use the `-assert success` option to enable reporting of successful matches, and successes on `cover` statements, in addition to failures.

- Limit the maximum number of successes reported

You can use the `-assert maxsuccesses=N` option to limit the total number of reported successes to N .

- Disable the display of messages when assertions fail

You can use the `-assert quiet` option to disable the display of messages when assertions fail.

- Enable or disable assertions during runtime
You can use the `-assert hier=file_name` option to enable or disable the list of assertions in the specified file.
- Generate a report file
You can use the `-assert report=file_name` option to generate a report file with the specified name. For additional information, see “[Options for SystemVerilog Assertions](#)” on page [B-9](#).

You can enter more than one keyword, using the plus + separator. For example:

```
% vcs -assert maxfail=10+maxsucess=20+success ...
```

However, you cannot combine the elaboration assert arguments and runtime assert arguments. Both should be specified separately as shown below:

```
% vcs -assert disable+dumpoff
      -assert maxfail=10+maxsucess=20+success ...
```

Assertion Monitoring System Tasks

For monitoring SystemVerilog assertions we have developed the following new system tasks:

```
$assert_monitor
$assert_monitor_off
$assert_monitor_on
```

Note:

Enter these system tasks in an initial block. Do not enter these system tasks in an always block.

The `$assert_monitor` system task is analogous to the standard `$monitor` system task in that it continually monitors specified assertions and displays what is happening with them (you can have it only display on the next clock of the assertion). The syntax is as follows:

```
$assert_monitor( [0|1,] assertion_identifier... ) ;
```

Where:

0

Specifies reporting on the assertion if it is active (VCS is checking for its properties) and for the rest of the simulation reporting on the assertion or assertions, whenever they start.

1

Specifies reporting on the assertion or assertions only once, the next time they start.

If you specify neither 0 or 1, the default is 0.

assertion_identifier...

A comma separated list of assertions. If one of these assertions is not declared in the module definition containing this system task, specify it by its hierarchical name.

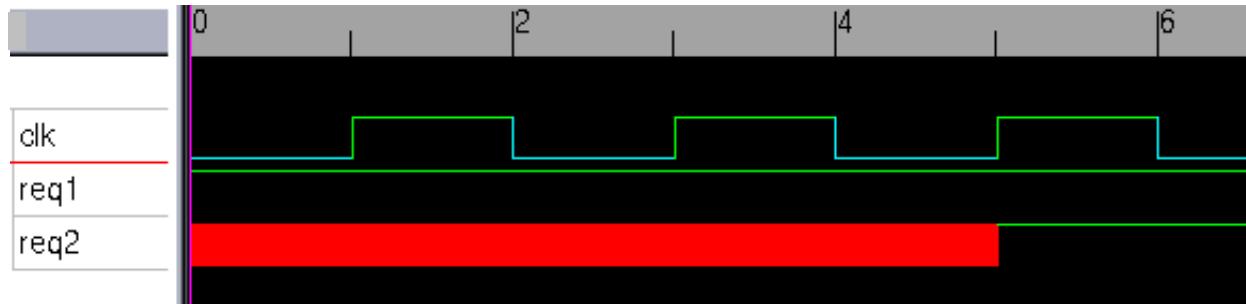
Consider the following assertion:

```
property p1;
  @ (posedge clk) (req1 ##[1:5] req2) ;
endproperty

a1: assert property(p1) ;
```

For property `p1` in assertion `a1`, a clock tick is a rising edge on signal `clk`. When there is a clock tick VCS checks to see if signal `req1` is true, and then to see if signal `req2` is true at any of the next five clock ticks.

In this example simulation, signal `clk` initializes to 0 and toggles every 1 ns, so the clock ticks at 1 ns, 3 ns, 5 ns and so on.



A typical display of this system task is as follows:

```
Assertion test.a1 ['design.v'27] :  
5ns: tracing "test.a1" started at 5ns:  
      attempt starting found: req1 looking for: req2 or  
      any  
5ns: tracing "test.a1" started at 3ns:  
      trace: req1 ##1 any looking for: req2 or any  
      failed: req1 ##1 req2  
5ns: tracing "test.a1" started at 1ns:  
      trace: req1 ##1 any[* 2 ] looking for: req2 or any  
      failed: req1 ##1 any ##1 req2
```

Breaking this display into smaller chunks:

```
Assertion test.a1 ['design.v'27] :
```

The display is about the assertion with the hierarchical name `test.a1`. It is in the source file named `design.v` and declared on line 27.

```
5ns: tracing "test.a1" started at 5ns:  
      attempt starting found: req1 looking for: req2 or  
      any
```

At simulation time, 5 ns VCS is tracing `test.a1`. An attempt at the assertion started at 5 ns. At this time, VCS found `req1` to be true and is looking to see if `req2` is true one to five clock ticks after 5 ns. Signal `req2` doesn't have to be true on the next clock tick, so `req2` not being true is okay on the next clock tick; that's what looking for "or any" means, anything else than `req2` being true.

```
5ns: tracing "test.a1" started at 3ns:  
      trace: req1 ##1 any looking for: req2 or any  
      failed: req1 ##1 req2
```

The attempt at the assertion also started at 3 ns. At that time, VCS found `req1` to be true at 3 ns and it is looking for `req2` to be true some time later. The assertion "failed" in that `req2` was not true one clock tick later. This is not a true failure of the assertion at 3 ns, it can still succeed in two more clock ticks, but it didn't succeed at 5 ns.

```
5ns: tracing "test.a1" started at 1ns:  
      trace: req1 ##1 any[* 2 ] looking for: req2 or any  
      failed: req1 ##1 any ##1 req2
```

The attempt at the assertion also started at 1 ns. [* is the repeat operator. `##1 any[* 2]` means that after one clock tick, anything can happen, repeated twice. So the second line here says that `req1` was true at 1 ns, anything happened after a clock tick after 1 ns (3 ns) and again after another clock tick (5 ns) and VCS is now looking

for `req2` to be true or anything else could happen. The third line here says the assertion “failed” two clock ticks (5 ns) after `req1` was found to be true at 1 ns.

The `$assert_monitor_off` and `$assert_monitor_on` system tasks turn off and on the display from the `$assert_monitor` system task, just like the `$monitoroff` and `$monitoron` system turn off and on the display from the `$monitor` system task.

Using Assertion Categories

You can categorize assertions and then enable and disable them by category. There are two ways to categorize SystemVerilog assertions:

- Using OpenVera assertions system tasks for categorizing assertions
- Using attributes

After you categorize assertions you can use these categories to stop and restart assertions.

Using OpenVera Assertion System Tasks

VCS has a number of system tasks and functions for OpenVera assertions that also work on SystemVerilog assertions. These system tasks do the following:

- Set a category for an assertion
- Return the category of an assertion

These system tasks are as follows:

```
$ova_set_category("assertion_full_hier_name",
    category)
```

or

```
$ova_set_category(assertion_full_hier_name,
    category)
```

System task that sets the category level attributes of an assertion.

The category level is an unsigned integer from 0 to $2^{24} - 1$.

Note:

These string arguments, such as the full hierarchical name of an assertion, can be enclosed in quotation marks or not. This is true when using these system tasks with SVA. They must be in quotation marks when using them with OVA.

```
$ova_get_category("assertion_full_hier_name")
```

or

```
$ova_get_category(assertion_full_hier_name)
```

System function that returns an unsigned integer for the category.

Using Attributes

You can prefix an attribute in front of an `assert` statement to specify the category of the assertion. The attribute must begin with the category name and specify an integer value, for example:

```
(* category=1 *) a1: assert property (p1);
(* category=2 *) a2: assert property (s1);
```

The value you specify can be an unsigned integer from 0 to $2^{24} - 1$, or a constant expression that evaluates to 0 to $2^{24} - 1$.

You can use a parameter, localparam, or genvar in these attributes. For example:

```
parameter p=1;
localparam l=2;
.
.
.
(* category=p+1 *) a1: assert property (p1);
(* category=l *) a2: assert property (s1);

genvar g;
generate
for (g=0; g<1; g=g+1)
begin:loop
(* category=g *) a3: assert property (s2);
end
endgenerate
```

Note:

In a generate statement the category value cannot be an expression, the attribute in the following example is invalid:

```
genvar g;
generate
for (g=0; g<1; g=g+1)
begin:loop
(* category=g+1 *) a3: assert property (s2);
end
endgenerate
```

If you use a parameter for a category value, the parameter value can be overwritten in a module instantiation statement.

You can use these attributes to assign categories to both named and unnamed assertions. For example:

```
(* category=p+1 *) a1: assert property (p1);  
(* category=1 *) assert property (s1);
```

The attribute is retained in a `tokens.v` file when you use the `-Xman=0x4` compile-time option and keyword argument.

Stopping and Restarting Assertions By Category

There are also OpenVera assertions system tasks for starting and stopping assertions that also work on SystemVerilog assertions. These system tasks are as follows:

`$ova_category_start(category)`

System task that starts all assertions associated with the specified *category*.

`$ova_category_stop(category)`

System task that stops all assertions associated with the specified *category*.

Using Mask Values To Stop And Restart Assertions

There are system tasks for both OpenVera and SystemVerilog assertions that allow you to use a mask to determine if a category of assertions should be stopped or restarted. These system tasks are `$ova_category_stop` and `$ova_category_start`. They have matching syntax.

```
$ova_category_stop(categoryValue,  
maskValue[,globalDirective]);
```

Where:

`categoryValue`

Because there is a `maskValue` argument, this argument is now the result of an anding operation between the assertion categories and the `maskValue` argument. If the result matches this value, these categories stop. As seen in “[Stopping and Restarting Assertions By Category](#)” on page 20-16, without the `maskValue` argument, this argument is the value you specified in `$ova_set_category` system tasks or `category` attribute.

`maskValue`

A value that is logically anded with the category of the assertion. If the result of this and operation matches the `categoryValue`, VCS stops monitoring the assertion.

`globalDirective`

Can be either of the following values:

0

Enables an `$ova_category_start` system task, that does not have a `globalDirective` argument, to restart the assertions stopped with this system task.

1

Prevents an `$ova_category_start` system task that does not have a `globalDirective` argument from restarting the assertions stopped with this system task.

```
$ova_category_start(categoryValue,  
                    maskValue[, globalDirective] );
```

Where:

`categoryValue`

Because there is a `maskValue` argument, this argument now is the result of an anding operation between the assertion categories and the `maskValue` argument. If the result matches this value, these categories start. As seen in “[Stopping and Restarting Assertions By Category](#)” on page 20-16, without the `maskValue` argument, this argument is the value you specified in `$ova_set_category` system tasks or `category` attribute.

`maskValue`

A value that is logically anded with the category of the assertion. If the result of this and operation matches the `categoryValue`, VCS starts monitoring the assertion.

`globalDirective`

Can be either of the following values:

0

Enables an `$ova_category_stop` system task, that does not have a `globalDirective` argument, to stop the assertions started with this system task.

1

Prevents an `$ova_category_stop` system task that does not have a `globalDirective` argument from stopping the assertions started with this system task.

Examples

This first example stops the odd numbered categories:

```
$sova_set_category(top.d1.a1,1);
$sova_set_category(top.d1.a2,2);
$sova_set_category(top.d1.a3,3);
$sova_set_category(top.d1.a4,4);

.
.
.
.

$sova_category_stop(1,'h1);
```

The categories are masked with the *maskValue* argument and compared with the *categoryValue* argument:

	bits	categoryValue	
category 1	001		
maskValue	1		
result	1	1	match
category 2	010		
maskValue	1		
result	0	1	no match
category 3	011		
maskValue	1		
result	1	1	match
category 4	100		
maskValue	1		
result	0	1	no match

1. VCS looks at the least significant bit of each category and logically ands that LSB to the *maskValue* argument, which is 1.

2. The results of these anding operations, 1 or true for categories 1 and 3, and 0 or false for categories 2 and 4, is compared to the *categoryValue*, which is 1, there is a match for categories 1 and 3.
3. VCS stops the odd numbered categories.

This additional example uses the *globalDirective* argument:

```
$ova_set_category(top.d1.a1,1);
$ova_set_category(top.d1.a2,2);
$ova_set_category(top.d1.a3,3);
$ova_set_category(top.d1.a4,4);
.
.
.
$ova_category_stop(1,'h1,0);
$ova_category_stop(0,'h1,1);
.
.
.
$ova_category_start(1,'h1);
$ova_category_start(0,'h1);
```

In this example:

1. The two `$ova_category_stop` system tasks first stop the odd numbered assertions and then the even numbered ones. The first `$ova_category_stop` system task has a *globalDirective* argument that is 0, the second has a *globalDirective* argument that is 1.
2. The first `$ova_category_start` system task can restart the odd numbered assertions, but the second `$ova_category_start` system task cannot start the even numbered assertions.

Viewing Results

By default, VCS reports only assertion of the failures. However, you can use the `-assert success` runtime option to report both pass and failures.

Assertion results can be viewed:

- Using a Report File
- Using DVE

For information on viewing assertions in DVE, refer to “[Working with Assertions and Cover Properties](#)”.

Using a Report File

Using the `-assert report=file_name` option, you can create an assertion report file. VCS writes all SVA messages to the specified file.

Assertion attempts generate messages with the following format:

File and line with the assertion	Full hierarchical name of the assertion	Start time	Status (succeeded at ..., failed at ..., not finished)
"design.v", 157: top.cnt_in.a2:	started at 22100ns	failed at 22700ns	

Offending ' (busData == mem[\$past(busAddr, 3)])'

Expression that failed (only with failure of check assertions)

21

Using Property Specification Language

VCS supports the Simple Subset of the IEEE 1850 Property Specification Language (PSL) standard. Refer to Section 4.4.4 of the *IEEE 1850 PSL LRM* for the subset definition.

You can use PSL along with SystemVerilog Assertions (SVA), SVA options, SVA system tasks, and OpenVera (OV) classes.

Including PSL in the Design

You can include PSL in your design in any of the following ways:

- Inlining the PSL using the `//psl` or `/*psl */` pragmas in Verilog and SystemVerilog.
 - Specifying the PSL in an external file using a verification unit (`vunit`).
-

Examples

The following example shows how to inline PSL in Verilog using the `//psl` and `/*psl */` pragmas.

```
module mod;
    ...
    // psl a1: assert always {r1; r2; r3} @(posedge clk);

    /* psl
       A2: assert always {a;b} @(posedge clk);
       ...
    */
endmodule
```

The following example shows how to use `vunit` to include PSL in the design.

```
vunit vunit1 (verilog_mod)
{
    a1: assert always {r1; r2; r3} @(posedge clk);
}
```

Usage Model

If you inline the PSL code, you must compile it with the `-psl` option.

If you use `vunit`, you must compile the file that contains the `vunit` with the `-pslfile` option. You do not need to use this option if the file has the `.psl` extension.

Compilation

```
% vcs -psl [vcs_options] Verilog_files
```

Simulation

```
% simv
```

Examples

To simulate the PSL code inlined in a Verilog file (`test.v`), execute the following commands:

```
% vcs -psl test.v
% simv
```

To simulate the `vunit` specified in an external file with the `.psl` extension (`checker.psl`), execute the following commands:

```
% vcs dev.v checker.psl
% simv
```

To simulate the `vunit` specified in an external file without the `.psl` extension (`checker.txt`), execute the following commands:

```
% vcs dev.v -pslfile checker.txt
% simv
```

To simulate both the PSL code inlined in a Verilog file (`test.v`), and the vunit specified in an external file (`checker.psl` or `checker.txt`), execute the following commands:

```
% vcs -psl -test.v checker.psl  
% simv
```

or

```
% vcs -psl -test.v -pslfile checker.txt  
% simv
```

Using SVA Options, SVA System Tasks, and OV Classes

VCS enables you to use all assertion options with SVA, PSL, and OVA. For example, to enable PSL coverage and debug assertions while compiling the PSL code, execute the following commands:

```
% vcs -psl -cm assert -debug -assert enable_diag test  
% simv -cm assert -assert success
```

For information on all assertion options, see Appendix B, Compile Time Options.

You can control PSL assertions in any of the following ways:

- Using the `$asserton`, `$assertoff`, or `$assertkill` SVA system tasks.
- Using NTB-OpenVera assert classes.

Note that VCS treats the `assume` PSL directive as the `assert` PSL directive.

Discovery Visual Environment (DVE) supports PSL assertions. The PSL assertion information displayed by VCS is similar to SystemVerilog assertions.

Limitations

The VCS implementation of PSL has the following limitations:

- VCS does not support binding `vunit` to an instance of a module or entity.
- VCS does not support `vunit` inheritance.
- VCS does not support the following data types in your PSL code -- shortreal, real, realtime, associative arrays, and dynamic arrays.
- VCS does not support the `next()` PSL function.
- VCS does not support the `union` operator and union expressions in your PSL code.
- Clock expressions have the following limitations:
 - You must include the `posedge` or `negedge` edge descriptors.
 - You must not include the `rose()` and `fell()` built-in functions.
 - You must not include endpoint instances.
- Endpoint declarations must have a clocked SERE with either a clock expression or default clock declaration.
- VCS does not support multi-clocked SEREs.

- VCS does not support the `%for` and `%if` macros.
- VCS supports only the `always` and `never` FL invariance operators in top-level properties. Ensure that you do not instantiate top-level properties in other properties.
- VCS does not support LTL operators in your PSL code.
- VCS does not support the `assume_guarantee`, `restrict`, and `restrict_guarantee` PSL directives.

22

Using SystemC

The VCS SystemC Co-simulation Interface enables VCS and the SystemC modeling environment to work together when simulating a system described in the Verilog and SystemC languages.

VCS contains a built-in SystemC simulator that is compatible with OSCI SystemC 2.0.1, OSCI SystemC 2.1v1, and OSCI SystemC 2.2 (IEEE 1666). By default, when you use the interface, VCS runs its own SystemC 2.2 simulator. No setup is necessary.

SystemC has been standardized in the IEEE 1666 standard format. Only SystemC 2.2 is fully compatible to this standard. We encourage you to migrate towards version SystemC 2.2. However, please note that there are differences between the three SystemC versions, and therefore, some porting effort may be necessary. In addition, it is important to note that support for SystemC 2.1v1 in VCS is deprecated and will be removed in the future.

You also have the option of installing the OSCI SystemC simulator and having VCS run it to co-simulate using the interface. See “[Using a Customized SystemC Installation](#)” on page 69.

With the interface, you can use the most appropriate modeling language for each part of the system, and verify the correctness of the design. For example, the VCS SystemC Co-simulation Interface allows you to:

- Use a SystemC module as a reference model for the Verilog RTL design under test in your testbench
- Verify a Verilog netlist after synthesis with the original SystemC testbench
- Write test benches in SystemC to check the correctness of Verilog designs
- Import legacy Verilog IP into a SystemC description
- Import third-party Verilog IP into a SystemC description
- Export SystemC IP into a Verilog environment when only a few of the design blocks are implemented in SystemC
- Use SystemC to provide stimulus to your design

The VCS /SystemC Co-simulation Interface creates the necessary infrastructure to co-simulate SystemC models with Verilog models. The infrastructure consists of the required build files and any generated wrapper or stimulus code. VCS writes these files in subdirectories in the `./csrc` directory. To use the interface, you don't need to do anything to these files.

During co-simulation, the VCS /SystemC Co-simulation Interface is responsible for:

- Synchronizing the SystemC kernel and VCS
- Exchanging data between the two environments

Note:

- There are examples of Verilog instantiated in SystemC and SystemC instantiated in Verilog in the `$VCS_HOME/doc/examples/systemc` directory.
- The interface supports the following compilers:
 - Linux: gcc 3.3.6, gcc 3.4.6, and gcc 4.2.2 compilers
 - Solaris: SC 8.0, and gcc 3.3.2
- The VCS / SystemC Co-simulation Interface supports 32-bit, as well as 64-bit (VCS flag `-full64`) simulation.
- The gcc 4.2.2, gcc 3.4.6, and gcc 3.3.6 compilers along with a matching set of GNU tools are available on the Synopsys FTP server for download. For more information, e-mail vcs_support@synopsys.com.

This chapter describes the following sections:

- “[Overview](#)”
- “[Verilog Design Containing Verilog Modules and SystemC Leaf Modules](#)”
- “[SystemC Designs Containing Verilog Modules](#)”
- “[SystemC Only Designs](#)”
- “[Considerations for Export DPI Tasks](#)”
- “[Specifying Runtime Options to the SystemC Simulation](#)”
- “[Using a Port Mapping File](#)”

- “Using a Data Type Mapping File”
- “Combining SystemC with Verilog Configurations”
- “Parameters”
- “Debugging Mixed Simulations Using DVE or UCLI”
- “Transaction Level Interface”
- “Delta-cycles”
- “Using a Customized SystemC Installation”
- “Using Posix threads or quickthreads”
- “Extensions”
- “Installing VG GNU Package”
- “Static and Dynamic Linking”
- “Limitations”

Overview

VCS /SystemC Co-simulation Interface supports the following topologies:

- Verilog designs containing SystemC modules
In this topology, you have a Verilog testbench and instances of SystemC and Verilog. You can also have many other SystemC modules in the design. To instantiate a SystemC module in your Verilog design, create a Verilog wrapper and instantiate the wrapper in your Verilog testbench. You can use the `syscan` utility to create a Verilog wrapper for your SystemC module. To see the usage model and an example, refer to the section entitled, “[Verilog Design Containing Verilog Modules and SystemC Leaf Modules](#)”.
- SystemC designs containing Verilog modules
In this topology, you have a SystemC testbench and instances of Verilog. You can also have many other SystemC modules in the design. To instantiate a Verilog design in your SystemC module, create a SystemC wrapper and instantiate the wrapper in your SystemC module. You can use the `vlogan` executable to create a SystemC wrapper for your Verilog design units. To see the usage model and an example, refer to the section entitled, “[SystemC Designs Containing Verilog Modules](#)”.

For information on limitations, see “[Limitations](#)”.

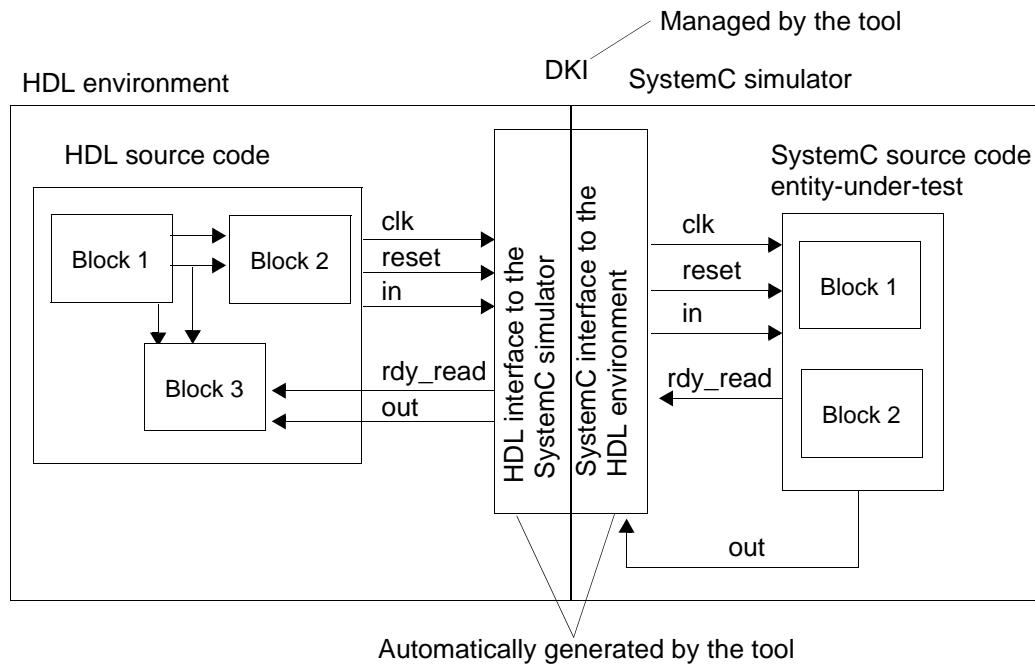
Verilog Design Containing Verilog Modules and SystemC Leaf Modules

To co-simulate a Verilog design that contains SystemC and Verilog modules, you need to create a Verilog wrapper for the SystemC module, which directly interacts with the Verilog design. You can instantiate your SystemC modules in the Verilog module just like instantiating any other Verilog module. For additional information,

see “[Example](#) on page 12”. The ports of the created Verilog wrapper are connected to signals that are attached to the ports of the corresponding SystemC modules.

[Figure 22-1](#) illustrates VCS DKI communication.

Figure 22-1 VCS DKI Communication of a Verilog Design Containing SystemC Modules



Usage Model

The usage model to simulate a design having a Verilog testbench with SystemC and Verilog instances involves the following steps:

1. Wrapper Generation
2. Compilation
3. Simulation

Wrapper Generation

```
% syscan [options] file1.cpp:sc_module_name
```

For additional information, see “[Generating Verilog Wrappers for SystemC Modules](#)”.

Compilation

```
% vcs -sysc [compile_options] file1.v file2.v
```

Simulation

```
% simv [runtime_options]
```

Input Files Required

To run a co-simulation with a Verilog design containing SystemC instances, you need to provide the following files:

- SystemC source code
 - You can directly write the entity-under-test source code or generate it with other tools
 - Any other C or C++ code for the design
- Verilog source code (.v extensions) including:
 - Verilog wrapper for your SystemC module (see “[Generating Verilog Wrappers for SystemC Modules](#)”)
 - Any other Verilog source files for the design
- An optional port mapping file. If you do not provide this file, the interface uses the default port mapping definition. For details of the port mapping file, see “[Using a Port Mapping File](#)” on page 37.

- An optional data type mapping file. If you don't write a data type mapping file, the interface uses the default one in the VCS installation. For details of the data type mapping files, see “[Using a Data Type Mapping File](#)” on page 38.

Generating Verilog Wrappers for SystemC Modules

You use the `syscan` utility to generate the wrapper and interface files for co-simulation. This utility creates the `csrc` directory in the current directory. The `syscan` utility writes the wrapper and interface files in subdirectories in the `./csrc` directory.

The syntax for the `syscan` command line is as follows:

```
syscan [options] filename[:modulename]
[filename[:modulename]]*
```

Where:

filename[:*modulename*] [*filename*[:*modulename*]]*

Specifies all the SystemC files in the design. There is no limit to the number of files.

Include `:modulename`, for those SystemC modules which are directly instantiated in your Verilog design. If `:modulename` is omitted, the `.cpp` files are compiled and added to the design's database so the final vcs command is able to bring together all the modules in the design. You do not need to add `-I$VCS_HOME/include` or `-I$SYSTEMC/include`.

[*options*]

These can be any of the following:

`-cflags "flags"`

Passes flags to the C++ compiler.

`-cpp path_to_the_compiler`

Specifies the location of the C++ compiler. If you do specify this option, VCS uses the following compilers by default:

- Linux : g++
- SunOS : CC (native Sun compiler)

Note:

- See the VCS *Release Notes* for details on all supported compiler versions.

`-full64`

Enables compilation and simulation in 64-bit mode.

`-debug_all`

Prepares SystemC source files for interactive debugging. Along with `-debug_all`, use the `-g` compiler flag.

`-port port_mapping_file`

Specifies a port mapping file. See “[Using a Port Mapping File on page 37](#)”.

`-Mdir=directory_path`

Specifies an alternate directory for 'csrc'.

`-help | -h`

Displays the syntax, options, and examples of the `syscan` command.

`-v`

Displays the version number.

`-o name`

The `syscan` utility uses the specified `name` instead of the module name as the name of the model. Do not enter this option when you have multiple modules on the command line. Doing so results in an error condition.

`-V`

Displays code generation and build details. Use this option if you encounter errors, or are interested in the flow that builds the design.

`-f filename`

Specifies a file containing one or more `filename [: modulename]` entries, as if these entries were on the command line.

`-verilog | -vhdl`

Generates wrapper for the specified language. `-verilog` is the default.

Note:

You do not specify the data type mapping file on the command line. For detailed information, see “[Using a Data Type Mapping File](#)” on page 38.

The following example generates a Verilog wrapper:

```
syscan -cflags "-g" sc_add.cpp:sc_add
```

Supported Port Data Types

SystemC types are restricted to the `sc_clock`, `sc_bit`, `sc_bv`, `sc_logic`, `sc_lv`, `sc_int`, `sc_uint`, `sc_bigint`, and `sc_bignum` data types. Native C/C++ types are restricted to the `uint`, `uchar`, `ushort`, `int`, `bool`, `short`, `char`, `long` and `ulong` types.

Verilog ports are restricted to `bit`, `bit-vector` and `signed bit-vector` types.

In-out ports that cross the co-simulation boundary between SystemC and Verilog must observe the following restrictions:

- SystemC port types must be `sc_inout_rv<>` or `sc_inout_resolved` and must be connected to signals of type `sc_signal_rv<>` or `sc_signal_resolved`.
- Verilog port types must be `bit_vector` or `bit`.
- You need to create a port mapping file, as described in “[Using a Port Mapping File](#)” on page 37, to specify the SystemC port data types as `sc_lv` (for a vector port) or `sc_logic` (for a scalar port).

Example

In this example, you have a Verilog testbench, a SystemC module, stimulus, and a Verilog module, display.

```
// SYSTEMC MODULE: stimulus
#include <systemc.h>
#include "stimulus.h"

void stimulus::entry() {

    cycle++;
    // sending some reset values
    if (cycle<25) {
        reset.write(SC_LOGIC_1);
        input_valid.write(SC_LOGIC_0);
    } else {
        reset.write(SC_LOGIC_0);
        input_valid.write( SC_LOGIC_0 );
        // sending normal mode values
        if (cycle%60==0) {
            input_valid.write(SC_LOGIC_1);
            sample.write( send_value1.to_int() );
            printf("Stimuli : %d\n", send_value1.to_int());
            send_value1++;
        }
    }
}
```

```

//Verilog module: display
module display (output_data_ready, result);
    input          output_data_ready;
    input [31:0] result;
    integer counter;

    ...

endmodule

//Verilog testbench: tb
module testbench ();

parameter PERIOD = 20;

reg clock;
wire reset;
wire input_valid;
wire [31:0] sample;
wire output_data_ready;
wire [31:0] result;

// Stimulus is the SystemC model.
stimulus stimulus1(.sample(sample),
                    .input_valid(input_valid),
                    .reset(reset),
                    .clk(clock));

// Display is the Verilog model.
display display1(.output_data_ready(output_data_ready),
                 .result(result));

...

endmodule

```

You can find the same example with a run script in the
\$VCS_HOME/doc/examples/systemc/ directory.

The usage model for the above example is shown below:

Wrapper Generation

```
% syscan stimulus.cpp:stimulus
```

For additional information, see “[Generating Verilog Wrappers for SystemC Modules](#)”.

Compilation

```
% vcs -sysc tb.v display.v
```

Simulation

```
% simv
```

Controlling Time Scale and Resolution in a SystemC

The SystemC runtime kernel has a time scale and time resolution that can be controlled by the user with functions

`sc_set_time_resolution()` and
`sc_set_default_time_unit()`. The default setting for time scale is 10 ns, default for time resolution is 10 ps.

The Verilog/VHDL runtime kernel also has a time scale and time resolution. This time scale/resolution is different and independent from the time scale/resolution of SystemC.

If the time scale/resolution is not identical, then a warning will be printed during the start of the simulation. The difference may slow down the simulation, may lead to wrong simulation results, or even make the simulation be "stuck" at one time point and not

progressing. It is therefore highly recommended to ensure that time scale and resolution from both kernels have the same settings. The following sections explain how to do this.

Automatic adjustment of the time resolution

When the time resolution of SystemC and HDL differs, the overall time resolution must be the finest of both. This can be set automatically by the elaboration option `-sysc=adjust_timeres` of vcs. This option determines the finest resolution used in both domains, and sets it to be the finest of the simulator. That can result that either the HDL side or the SystemC side is adjusted.

When it is not possible to adjust the time resolution, due to a user constraint, then an error is printed, and no simulator is created.

Setting time scale/resolution of Verilog/VHDL kernel

There are several ways how the time scale and resolution of a Verilog or mixed Verilog/VHDL is determined. For more information on time scale and resolution, see “[Controlling Time Scale and Resolution in a SystemC](#)” on page 14.

The most convenient way to ensure that Verilog/VHDL and SystemC use the same time scale/resolution is using the VCS `"-timescale=1ns/1ps"` command line option. Example:

```
vcs ... -sysc ... -timescale=1ns/1ps ...
```

This will force the Verilog/VHDL kernel to have the same values as the default values from the SystemC kernel. If this is not possible (for example, because you need a higher resolution in a Verilog module), then change the default values of the SystemC kernel as shown in the next section.

Setting time scale/resolution of SystemC kernel

The default time scale of a systemC kernel is 1 ns, and the default time resolution is 1 ps. These default values are NOT affected by the VCS -timescale option.

To control the time resolution of the SystemC kernel, create a static global object that initializes the timing requirements for the module. This can be a separate file that is included as one of the .cpp files for the design. Choose a value that matches the time scale/resolution of the Verilog/VHDL kernel.

The Sample contents for this file is as follows:

```
include <systemc.h>
class set_time_resolution {
public:
    set_time_resolution()
    {
        try {
            sc_set_time_resolution(10, SC_PS);
            sc_set_default_time_unit(100, SC_PS);
        }
        catch( const sc_exception& x ) {
            cerr << "setting time resolution/default time unit
                    failed: " <<
                    x.what() << endl;
        }
    }
};

static int SetTimeResolution()
{
    new set_time_resolution();
    return 42;
}

static int time_resolution_is_set = SetTimeResolution();
```

Adding a Main Routine for Verilog-On-Top Designs

Normally, a Verilog-on-top design doesn't contain a `sc_main()` function, since all SystemC instantiations are done within the Verilog modules. However, it is possible to add a main routine to perform several initializations for the SystemC side. The basic steps are as follows:

- Create a C++ source file which contains the main function (see example below).

Note:

Do not name this main function as `sc_main`.

- Add the registration function which takes care of the proper calling of the user-defined main routine
- Analyze the file, using `syscan user_main.cpp`. This will add the file to the design database. Note that there are no other options required to analyze this file.

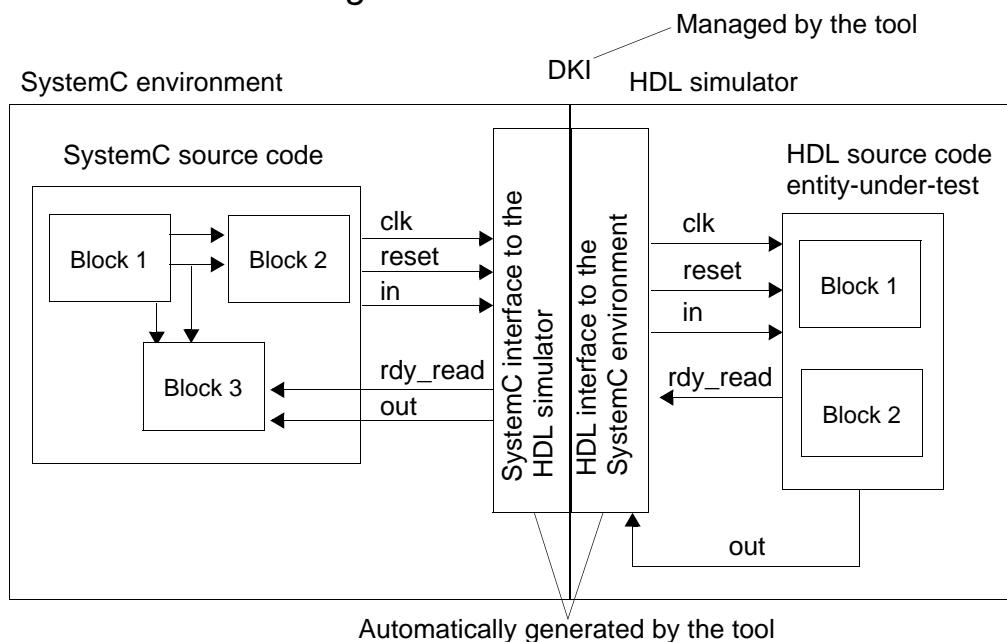
The user defined main routine must look like the following:

```
// File user_main.cpp
int user_main_function(int argc, char **argv)
{
    // you have access to the argc,argv arguments:
    for (int i = 0; i < (argc-1); ++i)
        std::cerr << Arg[" " << i << " ] = " << argv[i] << "\n";
    // do other init-stuff here...
    return 0;
}
extern "C" int sc_main_register(int (*)(int, char **));
static int my_sc_main =
sc_main_register(user_main_function);
// end-of user_main.cpp
```

SystemC Designs Containing Verilog Modules

To co-simulate a SystemC design that contains Verilog modules, you need to create header files for those Verilog instances which directly interact with the SystemC design. These header files will be named as `module_name.h` for Verilog modules (see “[Example](#) on page [22](#)”). You can analyze other Verilog files using the `vlogan` executable. The ports of the created SystemC wrapper are connected to signals that are attached to the ports of the corresponding Verilog modules.

Figure 22-2 VCS DKI Communication of SystemC Design Containing Verilog Modules



Usage Model

The usage model to simulate a design having a SystemC testbench with SystemC and Verilog instances involves the following steps:

1. Wrapper Generation
2. Compilation
3. Simulation

Wrapper Generation

```
% vlogan [options] -sysc -sc_model sc_module_name file1.v
```

For additional information, see “[Generating a SystemC Wrapper for Verilog Modules](#)”.

Compilation

```
% syscsim Verilog_files other_C++_source_files  
[compile_options]
```

Simulation

```
% simv [runtime_options]
```

Input Files Required

To run co-simulation with a SystemC design containing Verilog modules, you need to provide the following files:

- Verilog source code (.v extensions)
 - Verilog source files necessary for the design.
- SystemC source code including:
 - A SystemC top-level simulation (sc_main) that instantiates the interface wrappers and other SystemC modules.
 - Any other SystemC source files for the design.

- An optional port mapping file. If you do not provide this file, the interface uses the default port mapping definition. For details of the port mapping file, see “[Using a Port Mapping File](#)” on page 37.
- An optional data type mapping file. If you don’t write a data type mapping file, the interface uses the default file in the VCS installation. For details of the data type mapping files, see “[Using a Data Type Mapping File](#)” on page 38.

Generating a SystemC Wrapper for Verilog Modules

Use the `vlogan` utility with the `-sc_model` option to generate and build the wrapper and interface files for Verilog modules for co-simulation. This utility creates the `./csrc` directory in the current directory. The `vlogan` utility writes the header and interface files in the `./csrc/sysc/include` directory.

The syntax for the `vlogan` command line is as follows:

```
vlogan [options] -sc_model modulename file.v
```

Here the options are:

`-sc_model modulename file.v`

Specifies the module name and its Verilog source file.

`-cpp path_to_the_compiler`

Specifies the location of the C compiler. If you omit `-cpp path`, your environment will find the following compilers as defaults:

- Linux : g++
- SunOS : CC (native Sun compiler)

Note:

- See the *VCS Release Notes* for more details on supported compiler versions.
- You can override the default compilers in your environment by supplying a path to the g++ compiler. For example:

`-cpp /usr/bin/g++`

`-sc_portmap port_mapping_file`

Specifies a port mapping file. For additional information, see [“Using a Port Mapping File” on page 37](#).

`-Mdir=directory_path`

Works the same way that the `-Mdir` VCS compile-time option works. If you are using the `-Mdir` option with VCS, you should use the `-Mdir` option with vlogan to redirect the vlogan output to the same location that VCS uses.

`-V`

Displays code generation and build details. Use this option if you are encountering errors or are interested in the flow that builds the design.

For example, the following command line generates a SystemC wrapper and interface file for a Verilog module display:

```
vlogan -sc_model display display.v
```

Example

In this example, we have SystemC testbench `sc_main`, another SystemC module `stimulus`, and a Verilog module `display`.

```
// SystemC module: stimulus
#include <systemc.h>
#include "stimulus.h"

void stimulus::entry() {

    cycle++;
    // sending some reset values
    if (cycle<25) {
        reset.write(SC_LOGIC_1);
        input_valid.write(SC_LOGIC_0);
    } else {
        reset.write(SC_LOGIC_0);
        input_valid.write( SC_LOGIC_0 );
        // sending normal mode values
        if (cycle%60==0) {
            input_valid.write(SC_LOGIC_1);
            sample.write( send_value1.to_int() );
            send_value1++;
        };
    }
}

//Verilog module: display
module display (output_data_ready, result);
    input          output_data_ready;
    input [31:0] result;
    integer counter;

    ...

endmodule

//SystemC Testbench: sc_main
```

```

#include <systemc.h>
#include "stimulus.h"
#include "fir.h"
#include "display.h" //Header file for Verilog module display

int sc_main(int argc , char *argv[] ) {

    sc_clock clock ("CLK", 20, .5, 0.0);
    sc_signal<sc_logic>    reset;
    sc_signal<sc_logic>    input_valid;
    sc_signal<sc_lv<32> > sample;
    sc_signal<sc_logic>    output_data_ready;
    sc_signal<sc_lv<32> > result;

    display display1("display1" );
    stimulus stimulus1("stimulus1" );

    stimulus1.reset(reset);
    stimulus1.input_valid(input_valid);
    stimulus1.sample(sample);
    stimulus1.clk(clock.signal());
    fir1.reset(reset);
    fir1.input_valid(input_valid);
    fir1.sample(sample);
    fir1.output_data_ready(output_data_ready);
    fir1.result(result);
    fir1.clk(clock.signal());

    display1.output_data_ready(output_data_ready);
    display1.result(result);
    display1.input_valid(input_valid);
    display1.sample(sample);

    sc_start();
    return 0;
}

```

See \$VCS_HOME/doc/examples/systemc/ for examples.

The usage model for the above example is shown below:

Wrapper Generation

```
% vlogan -sysc -sc_model display display.v
```

For additional information, see “[Generating a SystemC Wrapper for Verilog Modules](#)” on page 20.

Compilation

```
% syscsim -sysc stimulus.cpp
```

Simulation

```
% simv
```

Compilation Scheme

When SystemC is at the top of the design hierarchy and you instantiate Verilog code in the SystemC code, the compilation of the simulation is done in the following two steps:

- The first step is to create a temporary simulation executable that contains all SystemC parts, but does not yet contain any HDL parts. VCS then starts this temporary executable to find out which Verilog instances are really needed. All SystemC constructors and `end_of_compilation()` methods are executed; however, simulation does not start.
- VCS creates the final version of the `simv` file containing SystemC, as well as all HDL parts. The design is now fully compiled and ready to simulate.

As a side effect of executing the temporary executable during step 1, you will see that the following message is printed:

```
Error- [SC-VCS-SYSC-ELAB] SystemC elaboration error
```

The design could not be fully elaborated due to an early exit of the SystemC part of the design. The SystemC part must execute the constructors of the design.

Please find the details in the SystemC chapter of the VCS documentation.

In case your simulation contains statements that should NOT be executed during step 1, guard these statements with a check for environment variable `SYSTEMC_ELAB_ONLY` or, with the following function:

```
extern "C" bool hdl_elaboration_only()
```

Both will be set/yield true only during this extra execution of `simv` during step 1.

For example, guard statements like this:

```
sc_main(int argc, char* argv[])
{
    // instantiate signals, modules, ...
    ModuleA my_top_module(...); // <-- must always be
                                 // executed

    // run simulation
    if (!hdl_elaboration_only()) {
        ... open log file for simulation report ...
    }
    sc_start();                  // <-- must always be executed
    if (!hdl_elaboration_only()) {
        ... close log file ...
    }

    return 0;
}
```

If you guard statements as mentioned above, make sure that all module constructors and at least one call of `sc_start()` will be executed.

VCS needs to know the entire SystemC module hierarchy during step 1, which in turn means that all SystemC module constructors must be executed.

If your simulation checks the command line arguments `argc + argv`, then you have two choices. Either guard these statements with an IF-statement as shown above.

Alternatively, provide the `simv` command line arguments used during elaboration using the VCS argument `-syscelab`. Example:

For non Unified Use Model (UUM) use model:

```
syscsim main.cpp ... -syscelab A ...
```

or, in UUM:

```
vcs -sysc sc_main ... -syscelab A ...
```

You can specify `-syscelab` multiple times. White space within the arguments is not preserved, instead the arguments are broken up into multiple arguments; multiple arguments can also be enclosed within double quotes, for example with `-syscelab "1 2 3"`.

If your SystemC design topology (the set of SystemC instances) depends on `simv` runtime arguments, then you MUST provide the relevant arguments with `-syscelab`. The SystemC design topology during step 1 and the final execution of `simv` must be identical.

Note that the `-syscelab` option is only supported when SystemC is at the top of the design hierarchy. If Verilog or VHDL is at the top, then `-syscelab` is neither needed nor supported.

SystemC Only Designs

VCS supports simulating and debugging simulations that contain only SystemC models, referred to as a "pure SystemC" simulation.

Pure SystemC simulations contain no Verilog, no SVA, and no NTB modules. The design will have only the SystemC and other C/C++ source files. The usage model to simulate pure SystemC designs is the same as SystemC on top designs, except the wrapper generation phase, which is not required for pure SystemC simulation.

Usage Model

The usage model to simulate a pure SystemC design involves the following steps:

1. Compilation
2. Simulation

Compilation

```
% syscan <SystemC source files(s)>
% vcs -sysc [compile_options] sc_main
```

Simulation

```
% simv [runtime_options]
```

Example 1:

```
% syscan adder.cpp  
% syscan foo.cpp bar.cpp xyz.cpp main.cpp  
% vcs -sysc sc_main  
% ./simv -gui
```

Example 2:

```
% syscan -cpp g++ -cflags -g adder.cpp  
% syscan -cpp g++ -cflags -g foo.cpp bar.cpp xyz.cpp  
% syscan -cpp g++ -cflags -g main.cpp  
% vcs -sysc sc_main \  
    -cpp g++ -cflags -g   \  
    extra_file.o -ldflags "-L/u/me/lib -labc"  
% ./simv -ucli
```

Restrictions

The following compilation options are not supported for pure SystemC simulation:

- sverilog: Pure SystemC simulation will not have any SV files.
- ntb*: Pure SystemC simulation will not have any OV files.
- ova*: Pure SystemC simulation will not have any OV files.
- cm*: Coverage related options are not supported.
- comp64: Cross-compilation is not supported. However, pure SystemC simulation is supported in 32-bit and 64-bit mode.
- e: The name of the main routine must always be `sc_main`.
- P: Pure SystemC simulation will not have any HDL files.

Supported and Unsupported UCLI/DVE and CBug Features

You can use UCLI commands or the DVE GUI to debug your pure SystemC design. The list of supported features in UCLI and DVE are as follows:

- View SystemC design hierarchy
- VPD tracing of SystemC objects
- Set breakpoints, stepping in C, C++, SystemC sources
- Get values of SystemC (or C/C++ objects)
- `stack [-up | -down]`
- `continue/step/next/finish`
- `run [time]`

The following UCLI and DVE features are not supported for SystemC objects:

- Viewing schematics
- Using force, release commands
- Tracing [active] drivers, and loads
- The UCLI command `next -end` is not supported.
- Commands that apply to HDL objects only

In case of a Control-C (i.e., SIGINT), CBug will always take over and report the current location.

When the simulation stops somewhere in the System C or VCS kernel, between execution of user processes, then a dummy file is reported as the current location. This happens, for example, immediately after the `init` phase. This dummy file contains a description about this situation and instructions how to proceed (i.e., Set BP in SystemC source file, click continue).

Controlling TimeScale Resolution

The most convenient way to ensure that Verilog and SystemC use the same time scale/resolution is using the VCS `-timescale=1ns/1ps` command-line option.

For example:

```
% vcs -sysc top.v -timescale=1ns/1ps
```

This command forces the Verilog kernel to have the same values as the default values from the SystemC kernel. If this is not possible (for example, because you need a higher resolution in a Verilog module), then change the default values of the SystemC kernel as shown in the following section.

Setting Timescale of SystemC Kernel

To control the time resolution of your SystemC module, create a static global object that initializes the timing requirements for the module. This can be a separate file that is included as one of the `.cpp` files for the design.

Sample contents for this file are:

```
include <systemc.h>
class set_time_resolution {
public:
    set_time_resolution()
    {
        try {
            sc_set_time_resolution(10, SC_PS);
        }
        catch( const sc_exception& x ) {
            cerr << "setting time resolution/default time unit
failed: " <<
x.what() << endl;
        }
    }
    static int SetTimeResolution()
    {
        new set_time_resolution();
        return 42;
    }
    static int time_resolution_is_set = SetTimeResolution();
```

Automatic Adjustment of Time Resolution

If the time resolution of SystemC and HDL differs, VCS can also automatically determine the finer time resolution and set it as the simulator's time scale. To enable this feature, you must use the `-sysc=adjust_timeres` compilation option.

VCS may be unable to adjust the time resolution if you elaborate your HDL with the `-timescale` option and/or use the `sc_set_time_resolution()` function call in your SystemC code. In such cases, VCS reports an error and does not create `simv`.

Considerations for Export DPI Tasks

If you have a SystemC design with Verilog instances, and you want to call export "DPI" tasks from the SystemC side of the design, then you need to do either one of the following three steps:

- “[Use syscan -export_DPI \[function-name\]](#)”
- “[Use syscan -export_DPI \[Verilog-file\]](#)”
- “[Use a Stubs File](#)”

Use syscan -export_DPI [*function-name*]

Register the name of all export DPI functions and tasks prior to the final `vcs` or `syscsim` call to compile the design. You need to call `syscan` in the following way:

```
syscan -export_DPI function-name1 [[function-name2] ...]
```

This is necessary for each export DPI task or function that is used by SystemC or C code. Only the name of function must be specified, and formal arguments are neither needed nor allowed. Multiple space-separated function names can be specified in one call of `syscan -export_DPI`. It is allowed to call `syscan -export_DPI` any number of times. A function name can be specified multiple times.

Example

Assume that you want to instantiate the following SystemVerilog module inside a SystemC module:

```
//myFile.v
module vlog_top;
    export "DPI" task task1;
    import "DPI" context task task2(input int A);
    export "DPI" function function3;

    task task1(int n);
        ...
    endtask
    function int function3(int m);
        ...
    endfunction // int
endmodule
```

You must do the following steps before you can compile the simulation:

```
syscan -export_DPI task1
syscan -export_DPI function3
```

Note that `task2` is not specified because it is an import "DPI" task.

Use `syscan -export_DPI [Verilog-file]`

This is same as `syscan -export_DPI [function-name]`, however, you can specify the name of a Verilog file instead of the name of an export DPI function. The `syscan` will search for all `export_DPI` declarations in that file.

The syntax is as shown below:

```
syscan -export_DPI [Verilog-file]
```

For example (see `myFile.v` in the above section):

```
% syscan -export_DPI myFile.v
```

This will locate `export_DPI` functions `task1` and `functions3` in the `myFile.v` file.

Note: `syscan` does not apply a complete Verilog or SystemVerilog parser, but instead does a primitive string search in the specified file.

The following restrictions apply:

- The entire `export_DPI` declaration must be written in one line (no line breaks allowed)
- ``include` statements are ignored
- Macros are ignored

VCS will compile the design even if the source files do not comply to the above restrictions. However, `syscan` will be unable to extract some or all of the `export_DPI` declarations. In this case, use `syscan -export_DPI [function-name]`.

Use a Stubs File

An alternative approach is to use stubs located in a library. For each export DPI function like `my_export_DPI`, create a C stub with no arguments and store it in an archive which is linked by VCS:

```
file my_DPI_stubs.c :  
    #include <stdio.h>  
    #include <stdlib.h>  
  
    void my_export_DPI() {  
        fprintf(stderr,"Error: stub for my_export_DPI is  
                used\n");  
        exit(1);  
    }  
  
    ... more stubs for other export DPI function ...  
  
gcc -c my_DPI_stubs.c  
ar r my_DPI_stubs.a my_DPI_stubs.o  
...  
syscsim ... my_DPI_stubs.a ...
```

It is important to use an archive (file extension `.a`) and not an object file (file extension `.o`).

Using options `-Mlib` and `-Mdir`

You can use VCS options `-Mlib` and `-Mdir` during analysis and elaboration to store analyzed SystemC files in multiple directories. This may be helpful if analyzing (compiling) of SystemC source files takes a long time, and if you want to share analyzed files between different projects.

The use model is as follows:

```
syscan -Mdir=<dir1> model1.cpp:model1
...
syscan -Mdir=<dir2> model2.cpp:model2
...
vcs -sysc -Mlib=<dir1>,<dir2> ...
```

Options `-Mlib` and `-Mdir` are available in all configurations, meaning for SystemC designs containing Verilog/VHDL modules, and also for Verilog/VHDL designs containing SystemC modules.

Specifying Runtime Options to the SystemC Simulation

You start a simulation with the `simv` command line. Command line arguments can be passed to just the VCS simulator kernel, or just the `sc_main()` function or both.

By default, all command-line arguments are given to `sc_main()`, as well as the VCS simulator kernel. All arguments following `-systemcrun` will go only to `sc_main()`. All arguments following `-verilogrun` will go only to the VCS simulator kernel. Argument `-ucli` is always recognized and goes only to the VCS simulator kernel.

For example:

```
simv a b -verilogrun c d -systemcrun e f -ucli g
```

Function `sc_main()` will receive arguments "a b e f g". The VCS simulator kernel will receive arguments "c d -ucli".

Using a Port Mapping File

You can provide an optional port mapping file for the `syscan` command with the `-port` option, and for `vlogan` by using `-sc_portmap`. If you specify a port mapping file, any module port that is not listed in the port mapping file is assigned the default type mapping.

A SystemC port has a corresponding Verilog port in the wrapper for instantiation. The `syscan` utility either uses the default method for determining the type of the HDL port it writes in the wrapper or uses the entry for the port in the port mapping file.

A port mapping file is an ASCII text file. Each line defines a port in the SystemC module, using the format in Example 14-1 and 14-2. A line beginning with a pound sign (#) is a comment.

A port definition line begins with a port name, which must be the same name as that of a port in the HDL module or entity. Specify the number of bits, the HDL port type, and the SystemC port type on the same line, separated by white space. You can specify the port definition lines in any order. You must, however, provide the port definition parameters within each line in this order: port name, bits, HDL type, and SystemC type.

The valid Verilog port types, which are case-insensitive, are as follows:

- `bit` — specifies a scalar (single bit) Verilog port
- `bit_vector` — specifies a vector (multi-bit) unsigned Verilog port (`bit-vector` is a valid alternative)

- `signed` — specifies a Verilog port that is also a reg or a net declared with the `signed` keyword and propagates a signed value.

The following example shows a port mapping file:

Example 22-1 Port Mapping File

#	Port name	Bits	Verilog type	SystemC type
	in1	8	<code>signed</code>	<code>sc_int</code>
	in2	8	<code>bit_vector</code>	<code>sc_lv</code>
	clock	1	<code>bit</code>	<code>sc_clock</code>
	out1	8	<code>bit_vector</code>	<code>sc_uint</code>
	out2	8	<code>bit_vector</code>	<code>sc_uint</code>

SystemC types are restricted to the `sc_clock`, `sc_bit`, `sc_bv`, `sc_logic`, `sc_lv`, `sc_int`, `sc_uint`, `sc_bigint`, and `sc_biguint` data types.

Native C/C++ types are restricted to the `bool`, `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, and `ulong` data types.

Using a Data Type Mapping File

When running a VCS / SystemC simulation, the interface propagates data through the module ports from one language domain to another. This can require the interface to translate data from one data type representation to another. This translation is called mapping, and is controlled by data type mapping files.

The data type mapping mechanism is similar to that used for port mapping, but is more economical and requires less effort to create and maintain. Because the data type mapping is independent of the

ports, you can create one or more default mappings for a particular type that will be used for all ports, rather than having to create a port map for every port of each new HDL wrapper model.

Data type mapping files map types, so that ALL ports of that type on ALL instances will now be assigned the specified mapping.

The data type mapping file is named `cosim_defaults.map`. The interface looks for and reads the data mapping file in the following places and in the following order:

1. In `$VCS_HOME/include/cosim`
2. In your `$HOME/.synopsys_ccss` directory
3. In the current directory.

An entry in a later file overrules an entry in an earlier file.

Each entry for a SystemC type has the following:

1. It begins with the keyword `Verilog`.
2. It is followed by the bit width. For vectors, an asterisk (*) is a wildcard to designate vectors of any bit width not specified elsewhere in the file.
3. The corresponding Verilog “type” using keywords that specify if it is scalar, unsigned vector, or signed port, the same keywords used in the port mapping file.
4. The SystemC or Native C++ type.

[Example 22-2](#) shows an example of a data type mapping file.

Example 22-2 Data Type Mapping File

```
#####
# Mappings between SystemC and Verilog datatypes
#####
Verilog * bit_vector      sc_bv
Verilog 1 bit              bool
Verilog * bit_vector      int
Verilog * signed           int
Verilog 1 bit              sc_logic
Verilog 1 bit              sc_bit
Verilog * bit_vector      char
Verilog * bit_vector      uchar
Verilog * bit_vector      short
Verilog * bit_vector      ushort
Verilog * bit_vector      uint
Verilog * bit_vector      long
Verilog * bit_vector      ulong
```

Combining SystemC with Verilog Configurations

SystemC can be used in combination with Verilog configurations. This is supported since release 2009.06 and only in UUM flow. Topologies SystemC-top and Verilog-top are supported. Topology VHDL-top is not (yet) supported.

Verilog-on-top, SystemC and/or VHDL down

A Verilog-on-top design with SystemC and/or VHDL down is specified like any other design, where the analysis of the Verilog files, by means of vlogan, must use the `libmap` option. Added to it is a Verilog source file, containing the configurations. A configuration consists of a config scope. Example:

```
config use_A;
    design top; // name of the Verilog top-entity
    default liblist workA; // library where the top-entity
                           // is analyzed
    // different mappings of verilog instances:
    instance top.v_mod.inst1 use workA.v_sub; // verilog-
                                              subtractor
    instance top.v_mod.inst2 use workA.h_sub; // VHDL-
                                              subtractor
    instance top.v_mod.inst3 use workA.s_sub; // SystemC-
                                              subtractor
endconfig

config use_B;
    design top;
    default liblist workA;
    // no overrule for ...inst1
    instance top.v_mod.inst2 use workA.s_sub; // SystemC-
                                              subtractor
    instance top.v_mod.inst3 use workA.s_sub; // SystemC-
                                              subtractor
endconfig
```

The name of the Verilog top-entity is obligatory. The default liblist statement defines where this Verilog top-entity is analyzed, by means of the `libmap` option of `vlogan`.

The instances are defined by their logical hierarchical name within the design hierarchy.

For setting up a design with Verilog configurations, it is required that, for the SystemC coupling, at least one module is a SystemC module. Later on this module can be configured to another module type, like Verilog or VHDL. That means, for a Verilog-on-top design, there must be at least one call to syscan like the one given below:

```
%> syscan s_sub.cpp:s_sub
```

that generates an interface model must be instantiated in Verilog.

The libmap option for vlogan requires a correct setting of the `synopsys_sim.setup` file. See the vcs and vcs-mx user guides for details.

Compiling a Verilog/SystemC design

Compiling a design containing only Verilog and SystemC is different compared to compiling a design containing Verilog, SystemC, and VHDL. Point of difference are the options passed to vcs for elaboration.

Note:

This can change in the future, when the `v2kconfig` is made default.

The following example shows how to compile a design containing only Verilog and SystemC:

```
%> syscan s_sub.cpp:s_sub -sysc=2.2
%> vlogan v_sub.v -libmap liblist.map -sverilog
%> vlogan v_design.v -libmap liblist.map -sverilog
%> vlogan v_configs.v -libmap liblist.map -sverilog
%> vcs -sverilog -top use_B -sysc=2.2
```

The used configuration for the design is specified with the option
"-top <config-name>".

Note:

Both options can change in the future once this flow is made default.

When a different configuration is to be used, or a configuration has changed, it is sufficient to re-analyze the verilog file containing the changed configuration, and redo the elaboration.

Compiling a Verilog/SystemC+VHDL design

Here an example how to compile a design:

```
%> syscan s_sub.cpp:s_sub -sysc=2.2
%> vlogan v_sub.v -lbimap liblist.map -sverilog
%> vhdlan h_sub.vhdl
%> vlogan v_design.v -libmap liblist.map -sverilog
%> vlogan v_configs.v -libmap liblist.map -sverilog
%> vcs -sysc=2.2 use_A -sverilog
```

Note the difference to the compile steps of SystemC+Verilog: the used configuration is NOT preceded with the -top option. Again, this may change in the future.

SystemC-on-top, Verilog and/or VHDL down

A SystemC-on-top design with Verilog and/or VHDL down is specified like any other design, where the analysis of the Verilog files, by means of vlogan, must use the libmap option. Added to it is a Verilog source file, containing the configurations. The following example shows a configuration with a SystemC-on-top topology:

```

config use_SysC_A;
    design sYsTeMcToP; // name of the default SystemC top
                        entity
    default liblist workA; // library where the top-entity
                          is analyzed
    // different mappings of verilog instances:
    instance sYsTeMcToP.v_mod.inst1 use workA.v_sub; // 
                                                verilog-subtractor
    instance sYsTeMcToP.v_mod.inst2 use workA.v_add; // 
                                                verilog-adder
    instance sYsTeMcToP.\sctop.sc2 .v_mod.inst3 use
                                                workA.v_add;
endconfig

config use_SysC_B;
    design sYsTeMcToP;
    default liblist workA;
    instance sYsTeMcToP.v_mod.inst1 use workA.h_sub; // 
                                                VHDL-subtractor
    instance sYsTeMcToP.v_mod.inst2 use workA.v_sub; // 
                                                verilog-subtractor
    instance sYsTeMcToP.\sctop.sc2 .v_mod.inst3 use
                                                workA.v_add;
endconfig

```

The name of the SystemC top-entity is hard coded as `sYsTeMcToP` and cannot be changed. Note that only Verilog modules can be re-configured; it is not possible to reconfigure a SystemC instance and/or a VHDL instance. Also note that it is not possible to re-configure a Verilog-instance to a SystemC-instance.

How to specify the pathname for a Verilog instance depends on the position of the instance within the design hierarchy.

Use a normal path for Verilog modules that are instantiated at the top-level inside the `sc_main()` function and that are not a sub-instance of a SystemC model. Example:

```
"instance sYsTeMcToP.v_mod.inst1"
```

But you must use a partially escaped path name for Verilog instances that are sub-instances of SystemC modules. The path name has to be split into two parts, where the first part contains only SystemC instances, and a second part contain Verilog/VHDL instances. The first part has to be specified as an extended Verilog identifier. Example:

```
instance sYsTeMcToP.\sctop.sc2 .v_mod.inst3 use  
                                workA.v_add;
```

The design topology is:

sctop	SystemC
sc2	SystemC
v_mod	Verilog
inst3	Verilog

The first part consists of two SystemC instances, 'sctop' and 'sc2'. These instances must be specified as "`\sctop.sc2`".

Note that the space at the end is important and must not be omitted. The second part consists of two Verilog instances, 'v_mod' and 'inst3' and must not be escaped.

Note that writing the configuration as

```
instance sYsTeMcToP.sctop.sc2.v_mod.inst3 use  
                                workA.v_add;
```

will not work at the moment, VCS will error out, but may be supported in future releases.

Compiling a SystemC/Verilog design

Compiling a design containing only Verilog and SystemC is different than compiling a design containing Verilog, SystemC, and VHDL. Point of difference are the options passed to vcs for elaboration.

Note:

This can change in the future, when the v2kconfig flow is made default.

```
%> vlogan v_sub.v -libmap liblist.map -sverilog  
%> vlogan v_mod.v -libmap liblist.map -sverilog -sc_model  
                                v_mod -sysc=2.2  
%> vlogan v_configs.v -libmap liblist.map -sverilog  
%> syscan sc_main.cpp -sysc=2.2  
%> vcs -sysc=2.2 -top use_A sc_main -sverilog
```

The used configuration is specified with the `-top <config-name>` option.

Note:

Both options can change in the future once this flow is made default. The argument `sc_main` specifies that the design topology is SystemC-on-top.

When a different configuration is to be used, or a configuration has changed, it is sufficient to re-analyze the verilog file containing the changed configuration, and redo the elaboration.

Compiling a SystemC/Verilog+VHDL design

```
%> vlogan v_sub.v -libmap liblist.map -sverilog  
%> vhdlan h_sub.vhdl  
%> vlogan v_mod.v -libmap liblist.map -sverilog -sc_model  
v_mod -sysc=2.2  
%> vlogan v_configs.v -libmap liblist.map -sverilog  
%> syscan -sysc=2.2 sc_main.cpp  
%> vcs -sysc=2.2 sc_main use_B -sverilog
```

Note:

The difference with MX-design is that the used configuration is NOT preceded with the `-top` option. Also, this may change in the future.

Limitations

The following limitation apply:

- VHDL-on-top designs are not supported with Verilog configurations.
- A Verilog-on-top design must contain at least one SystemC instance, when no configurations are used. Later on, this SystemC instance can be configured to something else.
- The name of the SystemC-top entity is hard coded to `sYsTeMcToP`. This may change in the future.
- The interfaces of the modules must match. The results are unpredicted otherwise. It is the user's responsibility to keep the consistence here.

The following options and implementation details may change in future releases:

- Some VCS elaboration options may change in future releases. Especially the options for the library map file, and the elaboration option `-top` can change.
- The name `sYsTeMCToP` to identify the root of a SystemC-top design may change in future releases.
- Use of partially escaped path names is currently necessary for certain instances of a SystemC-top topology. The requirement to escape some parts of the pathname may be removed in future releases.

Parameters

Parameters are supported between Verilog/VHDL and SystemC. The parameter values that are specified for a SystemC instance in Verilog are automatically passed to the SystemC domain.

Parameters in Verilog

Supported parameter types in Verilog are signed and unsigned integers and the real data type. For SystemVerilog, the string parameter type is also supported. Parameters are part of a module declaration and can be used like:

```
parameter msb = 7;
parameter e = 7, f = 5;
parameter foo = 8; bar = foo + 42;
parameter av_delay = (e + f) / 2;
parameter signed [3:0] mux_selector = 3;
parameter real pi = 314e-2;
parameter string hi_there = "Verilog String Parameter";
```

Parameters in VHDL

In VHDL, parameters correspond to 'generics'. Supported parameter types for the combination with Verilog and SystemC are integer, natural, real, and string. Generics are defined as part of the entity declaration:

```
entity H is
  generic (
    param_int : integer := 42;
    param_real : real := 123.456;
    param_nat : natural := 4;
    param_string : string := "VHDL String Parameter")
  port ( ... );
end H;
```

Parameters in SystemC

In SystemC, there is no standard definition for parameters. Therefore, a special parameter class is defined for that purpose. The supported types must match the types as being used in (System)Verilog and VHDL, so the supported datatypes are int, double and std::string. Within SystemC the parameters must be initialized with a default value inside the class constructor. Example:

```
#include "systemc.h"
#include "snps_hdl_param.h"

SC_MODULE(sysc_foo) {
    // declarative part
    hdl_param<int> msb;
    hdl_param<int> e, f;
    hdl_param<double> av_delay;
    hdl_param<std::string> hi_there;

    // initialization part
    SC_CTOR(sysc_foo) : HDL_PARAM(msb, 42), HDL_PARAM(e, 3),
        HDL_PARAM(f, 4), HDL_PARAM(av_delay, "123.456"),
        HDL_PARAM(hi_there, "SystemC String Parameter")
    { ... }
}
```

Verilog-on-Top, SystemC-down

The instantiation of a parameterized SystemC module inside Verilog is the same as for any other Verilog module:

```
sysc_foo #(11, 2, 3, 12.21, "Verilog-override") foo1(...);
sysc_foo #(.av_delay(44.33), .e(-9)) foo2(...);
sysc_foo foo3(...); // using all default parameter values
```

Within the SystemC constructor, the values of the parameters can be obtained by:

```
// SCCTOR(sysc_foo) : HDL_PARAM(...)...  
{  
    int l_msb = msb.get();  
    double delay = av_delay.get();  
    std::string str = hi_there.get();  
}
```

VHDL-on-Top, SystemC-down

The instantiation of a parameterized SystemC module inside VHDL is the same as for any other VHDL module:

```
architecture H_arch of H is  
    component sysc_foo  
        generic (  
            msb : integer;  
            e, f : integer;  
            av_delay : real;  
            hi_there : string )  
        port ( ... );  
    end component;  
  
begin  
    m_foo : sysc_foo generic map (  
        msb => 11;  
        av_delay => 0.01;  
        hi_there => "VHDL Override")  
    port map ( ... );  
    ...
```

SystemC-on-Top, Verilog/VHDL down

Within SystemC there are two ways to instantiate a foreign module:

- using the default constructor, and using separate setting calls for the parameters, or
- using a fully specified constructor, where each parameter must be assigned a value.

The instantiation can be in any SystemC module and/or in the `sc_main` routine:

```
#include "v_add.h" // verilog module
#include "h.h" // vhdl module
int sc_main(int, char **)
{
    h m_h("h"); // VHDL module
    m_h.param_int(44);
    m_h.param_real(99.01);
    m_h.param_string("SystemC Override");

    v_add m_v1("v1", 3 /* incr value */, 1.01 /* factor */,
               "SystemC Override");
    v_add m_v2("v2");
    m_v2.incr_value(4);
    m_v2.factor(0.99);

    m_v2.hi_there("SystemC Override #2");

    sc_start(-1, SC_NS);
}
```

The hdl_param class defines the ::operator() to initialize the parameters, and the ::get () function for obtaining the final value of the parameter. Parameters can only be initialized once, and cannot be altered after the value of the parameter is obtained by means of the ::get () function.

Namespace

For SystemC-2.1v1 and SystemC-2.2, name spaces are used to define the SystemC hdl_param objects:

```
namespace sc_snps {
    template < class T >
    class hdl_param : public sc_object { ... };
} // namespace sc_snps
```

For the declaration of the parameters this namespace must be used:

```
SC_MODULE(sysc_foo) {
    sc_snps::hdl_param<int> i;
};
```

Parameter specification as vcs elaboration arguments

Parameter can be defined using the vcs elaboration command line arguments. This is implemented only for a Verilog-on-top design:

```
* -pvalue+v_top.foo1.msb=33
```

This works only for integer and real parameter types. This doesn't work for string parameters.

```
* -parameters param.lst
```

with param.lst a list of parameter assignments (see the specific vcs part of this guide discussing parameters).

Debug

The SystemC hdl_param objects are visible as class parameters within a combined hierarchy view (vpd-file). Although parameters are constant and won't change after time == 0, they can be traced.

Access with the UCLI 'get' command is supported. Changing the value with the 'change' or 'force' commands is not supported, since parameters are constant after the construction time.

Limitations

The verilog parameters are not compile constants for SystemC. That has a limitation that these can not be used as template arguments for the construction of templated classes. Example:

```
SC_CTOR(not_possible) : HDL_PARAM(width, 4) {
    sc_int<width> *pint = new sc_int<width>; // NOT SUPPORTED
}
```

The same hold for the Verilog and VHDL domains:

```
module test( data_in1, data_in2 );
    parameter width = 12;
    input [width-1 : 0] data_in1; // NOT SUPPORTED
    input [11 : 0] data_in2;
endmodule
```

Debugging Mixed Simulations Using DVE or UCLI

You can use Discovery Visual Environment (DVE) or the Unified Command-line Interface (UCLI) to debug VCS simulations containing SystemC source code by attaching the C-source debugger to DVE or UCLI.

The following steps outline the general debugging flow. For more information, see *The Discovery Visual Environment User Guide* and the *Unified Command-line Interface User Guide*.

1. Compile your VCS with SystemC modules as you normally would, making sure to compile all SystemC files you want to debug.

For example, with a design with Verilog on top of a SystemC model:

```
% syscan -cpp g++ -cflags "-g" my_module.cpp:my_module  
% vcs -cpp g++ -sysc -debug_all top.v
```

Note that you must use `-debug` or `-debug_all` to enable debugging.

2. Start the debugger.

- To start DVE, enter:

```
simv -gui
```

- To start UCLI, enter:

```
simv -ucli
```

3. Attach the C debugger as follows:

- In DVE, select **Simulator > C/C++ Debugging** or enter `cbug` on the console command line.

In UCLI, enter `cbug` on the command line. Debugging SystemC source code is enabled and the following message appears:

```
CBug - Copyright Synopsys Inc. 2003-2009.
```

4. Run the simulation.

Transaction Level Interface

The transaction level interface (TLI) between SystemVerilog and SystemC supports communication between these languages at the transaction level. At RTL, all communication goes through signals. At transaction level, communication goes through function or task calls.

It is an easy-to-use feature that enables integrating Transaction Level SystemC models into a SystemVerilog environment seamlessly and efficiently. The automated generation of the communication code alleviates the difficulties in implementing a synchronized communication mechanism to fully integrate cycle accurate SystemC models into a SystemVerilog environment.

TLI exploits using the powerful Verification Methodology Manual (VMM methodology) to verify functional or highly accurate SystemC TLMs. TLI improves mixed language simulation performance and speeds-up the development of the verification scenarios. Furthermore, TLI adds the necessary logic to enable you to debug the transaction traffic using the waveform viewer in DVE.

TLI augments the pin-level interface (DKI) to enable both languages to communicate at different levels of abstraction. Using this interface, you can simulate some part of the design at the transaction-level and the other part at the hardware level, enabling full control over the level of detail required for your simulation runs. This integration also helps you to leverage the powerful features of SystemVerilog for transaction-level verification. Also, you can use the same testbenches for hardware verification. TLI enables you to do the following:

- Call interface methods of SystemC interfaces from SystemVerilog
- Call tasks or functions of SystemVerilog interfaces from SystemC

Methods and tasks can be blocking as well as non-blocking. Blocking in the context of this document means the call may not return immediately, but consumes simulation time before it returns. However, non-blocking calls always return immediately in the same simulation time.

The caller's execution is resumed exactly at the simulation time when the callee returns, so a blocking call consumes the same amount of time in both the language domains – SystemC and SystemVerilog. Non-blocking calls always return immediately.

The tasks or functions must be reachable through an interface of the specific language domain. This means that for SystemVerilog calling SystemC, the TLI can connect to functions that are members of a SystemC interface class. For SystemC calling SystemVerilog, the TLI can call functions or tasks that are part of a SystemVerilog interface.

The usage model of the transaction level interface consists of defining the interface by means of an interface definition file, calling a code generator to create the TLI adapters for each domain, and finally instantiation and binding of the adapters.

Interface Definition File

The interface definition file contains all the necessary information to generate the TLI adapters. It consists of a general section and a section specific to task/function. The order of lines within the general section is arbitrary, and the first occurrence of a task or function keyword marks the end of this section. The format of the file is illustrated as follows:

```
interface if_name
direction sv_calls_sc
[verilog_adapter name]
[systemc_adapter name]
[hdl_path XMR-path]

[#include "file1.h"]
[`include "file2.v"]
...

task <method1>
input|output|inout|return vlog_type argument_name_1 return
input|output|inout|vlog_type argument_name_2
.
.
.

function [return type] method2
input|output|inout vlog_type argument_name_1
.
.
.
```

The `interface` entry defines the name of the interface. For the direction SystemVerilog calling SystemC, the `if_name` argument must match the name of the SystemC interface class. Specialized template arguments are allowed in this case, for example `my_interface int` or `my_interface 32`. For SystemC calling SystemVerilog, `if_name` must match the SystemVerilog interface name.

The `direction` field specifies the caller and callee language domains, and defaults to `sv_calls_sc`. The SystemC calling SystemVerilog direction is indicated by `sc_calls_sv`.

The `verilog_adapter` and `systemc_adapter` fields are optional and define the names of the generated TLI adapters and the corresponding file names. File extension `.sv` is used for the `verilog_adapter` and file extensions `.h` and `.cpp` for the `systemc_adapter`.

The optional `#include` lines are inserted literally into the generated SystemC header file, and the optional ``include` lines into the generated SystemVerilog file.

The `hdl_path` field is optional and binds the generated Verilog adapter through an XMR to a fixed Verilog module, Verilog interface, or class instance. Using `hdl_path` makes it easier to connect to a specific entity, however, the adapter can be instantiated only once, not multiple times. If you want to have multiple connections, then create multiple adapters which differ only by their name.

A SystemC method may or may not be blocking, meaning it may consume simulation time before it returns or it will return right away. This distinction is important for the generation of the adapter. Use

task for SystemC methods that are blocking or even potentially blocking. Use function for SystemC methods that will not block for sure. Note that functions enable faster simulation than tasks.

The lines after task or function define the formal arguments of the interface method. This is done in SystemVerilog syntax. This means that types of the arguments must be valid SystemVerilog types. See “[Supported Data Types of Formal Arguments](#)” on page 66 for more details.

The return keyword is only allowed once for each task. It becomes an output argument on the Verilog side to a return value on the SystemC side. This feature is required because blocking functions in SystemC may return values, while Verilog tasks do not have a return value.

The one exception is if the methods of the SystemC interface class use reference parameters. For example, if my_method(int& par) is used, then you need to mark this parameter as inout& in the interface definition file. Note that the & appendix is only allowed for inout parameters. For input parameters, this special marker is not needed and not supported. Pure output parameters that should be passed as reference must be defined as inout in the interface definition file.

Example interface definition file for the simple_bus blocking interface:

```
interface simple_bus_blocking_if
direction sv_calls_sc
verilog_adapter simple_bus_blocking_if_adapter
```

```

systemc_adapter simple_bus_blocking_if_adapter
#include "simple_bus_blocking_if.h"

task burst_read
    input int unsigned priority_
    inout int data[32]
    input int unsigned start_address
    input int unsigned length
    input int unsigned lock
    return int unsigned status

task burst_write
    input int unsigned priority_
    inout int data[32]
    input int unsigned start_address
    input int unsigned length
    input int unsigned lock
    return int unsigned status

```

Generation of the TLI Adapters

The following command generates SystemVerilog and SystemC source code for the TLI adapters from the specified interface definition file:

```
syscan -tli_gen interface_definition_file
```

or

```
syscan -tli_gen_class interface_definition_file
```

This command generates SystemC and SystemVerilog files that define the TLI adapters for each language domain. All generated files can be compiled just like any other source file for the corresponding domain. The files have to be generated again only when the content of the interface definition file changes.

TLI adapters for the `sv_calls_sc` direction can be generated in two different styles. The SystemC part of the generate adapter is the same for both styles, however, the SystemVerilog part is different. Option `-tli_gen` creates a SystemVerilog "interface". Option `-tli_gen_class` creates a SystemVerilog "class". Both styles have benefits and penalties.

A class is generally easier to connect into the SystemVerilog source code and there are situations where a SystemVerilog testbench allows you to instantiate a class but not an interface. However, if a class is generated, then the TLI adapter can create only one connection of this type between the SystemVerilog and SystemC side. Alternatively, if an interface is generated, then multiple connections can be created (which are distinguished by the integer parameter of the interface).

Transaction Debug Output

Since the transaction information traveling back and forth between SystemVerilog and SystemC along with the transaction timing is often crucial information (for example, comparison of ref-model and design for debugging and so on), the SystemC part of the TLI adapters are generated with additional debugging output that can be enabled or disabled. For additional information, see [“Instantiation and Binding” on page 63](#).

The transaction debug output can either be used as a terminal I/O (`stdout`) or as a transaction tracing in DVE. In DVE, each TLI adapter has an `sc_signal<string>` member with name `m_task_or_function_name_transactions` that you can display in the waveform viewer of DVE.

Sometimes, the next transaction begins at the same point in time when the previous transaction ends. Prefixes ">" and "<" are used such that both transactions could be distinguished. The return values, if any, for the previous transaction are displayed with a leading "<". The input arguments for the new argument are prefixed with ">".

If the default scheme how the debug output is formatted does not match the debugging requirements, then do not change the generated code in the TLI adapter. Instead, override the debug methods `m_task_or_function_name_transactions` using a derived class that defines only these virtual methods. You can copy these methods from the generated adapter code as a starting point and then modify the code according to the debugging requirements.

If the adapter is generated again, then the existing code is overwritten and all manual edits are lost.

Note:

Do not manually modify the code generated by the TLI adapter. Instead, override the debug functions in a derived class.

Instantiation and Binding

TLI adapters must always be instantiated in pairs, where each pair forms a point-to-point connection from one language domain to the other.

If multiple pairs of the same TLI adapter type are needed in the design, you must instantiate the adapter multiple times in each domain. The point-to-point connection must be set up by assigning a matching ID value to the SystemVerilog interface or class, and the SystemC module. The ID value is set for SystemC module and the

SystemVerilog class, if generated, as a constructor argument. In case the SystemVerilog Adapter is generated as an interface, the ID is set through a parameter.

The SystemVerilog TLI adapter (either as an interface or a class) can be instantiated and used like any other SystemVerilog interface or class. If you want to call an IMC of a SystemC interface, you need to call the corresponding member function/task of the TLI adapter.

The SystemC part of the TLI adapter is a plain SystemC module that has a port `p` over the specified interface name (`sc_port if_name p`). This module can be instantiated in the systemC design hierarchy, where you can bind the port to the design interface just like any other SystemC module.

As mentioned above, there is an optional constructor argument for the point-to-point ID of type `int` that defaults to zero. There is a second optional constructor argument of type `int` that specifies the format of debug information that the adapter prints when an interface method is called. If the LSB of this argument is set, the TLI adapter prints messages to `stdout`. If the next bit (`LSB+1`) is set, this information is written to an `sc_signal<string>` that you can display in DVE.

For SystemC calling SystemVerilog, the SystemC part of the TLI adapter is an `sc_module` that you can instantiate within the module where you want to call the Verilog tasks or functions. You can execute the cross-boundary task or function calls by calling the corresponding member function of the SystemC TLI adapter instance.

The SystemVerilog portion of the TLI adapter depends on whether the `hdl_path` field is used and options `-tli_gen` or `-tli_gen_class` is used:

- combination -tli_gen, no hdl_path:
 The Verilog adapter has a port over the interface type, as defined in the interface description file. You can instantiate the adapter module in the Verilog design like any other Verilog module, and the port should be bound to the SystemVerilog interface that implements the tasks or functions to be called.
- combination -tli_gen, with hdl_path *path*:
 The Verilog adapter is a Verilog module with no ports. All calls initiated by SystemC are routed through the XMR path to some other Verilog module or interface.
- combination -tli_gen_class, with hdl_path *path*:
 The Verilog adapter is a group of task definitions and other statements that must be included in a program with an ``include "if_name_sc_calls_sv.sv"` statement. Calls initiated by the SystemC side are routed through the XMR path to some class object of the SV testbench.
- combination -tli_gen_class, no hdl-path:
 This combination is not supported and displays an error message.

It is important to note that Verilog tasks, in contrast to Verilog functions, must always be called from within a SystemC thread context. This is because tasks can consume time, and in order to synchronize the simulator kernels, `wait()` is used in the SystemC adapter module. The SystemC kernel throws an error when `wait()` is called from a non-thread context.

Supported Data Types of Formal Arguments

The TLI infrastructure uses the SystemVerilog DPI mechanism to call the functions and transport data, so the basic type mapping rules are inherited from this interface. Refer to the SystemVerilog standard for a detailed description on DPI. In summary, the following mapping rules apply for simple data types:

SystemVerilog	SystemC
input	byte
inout output	byte
input	shortint
inout output	shortint
input	int
inout output	int
input	longint
inout output	longint
input	real
inout output	real
input	shortreal
inout output	shortreal
input	chandle
inout output	chandle
input	string
inout output	string
input	bit
inout output	bit
input	logic
inout output	logic

For the integral data types in the above table, the signed and unsigned qualifiers are allowed and mapped to the equivalent C unsigned data type.

All array types listed in the above table are passed as pointers to the specific data types. There are two exceptions to this rule:

- Open arrays, which are only allowed for the SystemVerilog calling SystemC direction, are passed using handles (`void *`). The SystemVerilog standard defines the rules for accessing the data within these open arrays.
- Packed bit arrays with sizes ≤ 32 in input direction (for example, `input bit [31:0] myarg`) are passed by value of type `svBitVec32`. Basically, this type is an `unsigned int`, and the individual bits can be accessed by proper masking.

Miscellaneous

The TLI generator uses Perl5 which needs to be installed on the local host machine. Perl5 is picked up in the following order from your installation paths (1=highest priority):

1. `use ${SYSCAN_PERL}, if (defined)`
2. `/usr/local/bin/perl5`
3. `perl5` from local path and print warning

Delta-cycles

VPD dumping of delta-cycles is supported for SystemC elements, but it needs to be enabled as follows:

First, add function call `bf_delta_trace(1)` to the source code.
Example:

```
#include "cosim/bf/systemc_user.h"
...
int prev_state = bf_delta_trace(1);
```

This function turns on the delta tracing (or, turns off when the argument is 0). This function can be called anywhere, for example in constructors of SystemC classes, and/or in `sc_main`.

Next, make the generated delta-cycles visible in the DVE waveform window as follows:

1. Start the simulator with `-gui` option. This will pop up DVE.
2. Enable CBug Debugger in the DVE, and then select **Simulator -> C/C++ Debugging -> enable**.
Or, enter CBug in the DVE gui console command line.
3. Select **Simulator -> Capture Delta Cycle Values**. This will turn it on for DVE.
4. Go with the time-marker somewhere, and then Press right-mouse button.
5. select **Expand Time**.

Now the SystemC delta cycles are shown.

Using a Customized SystemC Installation

You can install the OSCI SystemC simulator 2.0.1, 2.1v1, or 2.2.0 and tell VCS to use it for Verilog/SystemC co-simulation. To do so, you need to:

- Obtain OSCI SystemC version from www.systemc.org.
- Set the `SYSTEMC` environment variable to path of the OSCI SystemC installation. For example:

```
setenv SYSTEMC /net/user/download/systemc-2.2.0
```

To create a SystemC 2.x installation with VCS patches, perform the following series of tasks.

There are several files in the `$VCS_HOME/etc/systemc-2.x` directories that contain necessary patches. You need to replace all SystemC files from the OSCI installation (*) with those from `$VCS_HOME/etc/systemc-2x2`.

(*): here is the location where you need to replace these files with the those from `$VCS_HOME`

For SC 2.2 and 2.1: `osci_SC_installation_path/src/sysc/kernel`

for SC 2.0.1: `osci_SC_installation_path/src/systemc/kernel`

For example, replace:

```
<your osci SystemC installation>/src/sysc/kernel/  
sc_simcontext.cpp
```

with:

```
$VCS_HOME/etc/systemc-2.2/sc_simcontext.cpp
```

Follow the installation instructions provided by OSCI (see file INSTALL which is part the SystemC tar file) and build a SystemC library. Note that you must use `../configure i686-pc-linux-gnu` to build a 32-bit Linux installation; call `../configure` on other platforms.

Set the `SYSTEMC_OVERRIDE` VCS environment variable to the user-defined OSCI SystemC library installation path. For example:

```
setenv SYSTEMC_OVERRIDE /net/user/systemc-2.2.0
```

Header files must exist in the `$SYSTEMC_OVERRIDE/include` directory and the `libsystemc.a` library file must be in the following directories:

- `$SYSTEMC_OVERRIDE/lib-linux/`
- `$SYSTEMC_OVERRIDE/lib-gccsparcos5/`

The `$SYSTEMC_OVERRIDE` environment variable must point to the OSCI SystemC simulator installation. Header files must be located at `$SYSTEMC_OVERRIDE/include` and library files in:

- `$SYSTEMC_OVERRIDE/lib-linux/libsystemc.a`
- `$SYSTEMC_OVERRIDE/lib-gccsparcos5/libsystemc.a`

Note that VcsSystemC 2.0.1 is binary compatible with OSCI SystemC 2.0.1.

As of July 2005, VcsSystemC 2.1 is binary compatible with OSCI SystemC 2.1v1.

As of March 19, 2007 (SYSTEMC_VERSION 20070314),
VcsSystemC 2.2 is binary compatible with OSCI SystemC 2.2.0.

Compatibility with OSCI SystemC

The default, built-in SystemC simulator is binary compatible to the OSCI SystemC 2.2.0 simulator. This means that you can link the object files (*.o, *.a, *.so) compiled with the OSCI SystemC 2.2.0 simulator to a `simv` executable without adding any switch to `vcs` or `syscan`.

The built-in SystemC simulator is also compatible with SystemC 2.1v1 and SystemC 2.0.1. To select, for example, SystemC 2.2, add `-sysc=2.0.1` as a command-line option during both analysis and compilation. If the `-sysc=<version>` option is not specified, VCS selects the default simulator which is SystemC-2.2.

Selecting SystemC 2.0.1, 2.1 or 2.2

VCS allows you to choose which version of SystemC by using the `-sysc=version` option. The default is 2.2. This option should be specified with `vhdlan`, `vlogan`, `vcs` and `syscim` as shown in the following example:

For example, to select version 2.2, use the `-sysc=2.2` option with `syscan`, `vhdlan`, `vlogan`, `vcs` and `syscim`.

```
% syscan -sysc=2.2 adder.cpp  
% syscan -sysc=2.2 adder.cpp:adder  
% vlogan -sysc=2.2 -sc_inst sub sub.v  
% vhdlan -sysc=2.2 -sc_inst mult mult.vhd  
% syscim -sysc=2.2 -sysc ...
```

The `-sysc` option accepts arguments `2.0.1`, `2.1` and `2.2`.

Again, if the `-sysc=version` option is not specified, then the default version `2.2` is selected.

The SystemC version must be consistent to all calls of `syscan`/`vlogan`/`vhdlan`/`vcs`/`syscsim` that go into the same database (`csrc` directory). If you try to mix different versions, VCS issues an error message during elaboration. For example, the following is not supported:

```
% syscan -sysc=2.2 adder.cpp  
% vlogan -sysc=2.1 -sc_inst sub sub.v  
% syscsim -sysc=2.2 -sysc ...
```

Compiling Source Files

If you need to compile the source files that include `systemc.h` in your own environment and not with the `syscan` script, then add compiler flag `-I$VCS_HOME/include/systemc22`, `-I$VCS_HOME/include/systemc21`, or `-I$VCS_HOME/include/systemc201` for SystemC 2.2, SystemC2.1v1, and SystemC 2.0.1 respectively.

Using Posix threads or quickthreads

This feature is only available with SystemC version 2.2.

SC_THREAD processes can be implemented by pthreads (Posix threads) or quickthreads. Switching from one SC_THREAD to another is significantly slower with pthreads than with quickthreads. However, pthreads have advantages in terms of debugging support with gdb or DVE/CBug or tools like Purify or Valgrind.

Whether pthreads or quickthreads are used depends on the platform and can be influenced by the user in some case(s).

- Linux 32-bit: always quickthreads
- Linux 64-bit: quickthreads are default, pthreads can be selected
- Solaris 32-bit: always quickthreads
- Solaris 64-bit: always pthreads

The following API allows you to select or check if pthreads are used (if supported on the platform):

```
// wish for either pthreads or quickthreads, return true  
// if wish is granted, return false+produce warning if not  
// possible.  
  
bool sc_snps::request_to_use_pthreads(bool use_pthreads);  
  
// use pthreads (true) or quickthreads for SC_[C] THREADS  
  
bool sc_snps::use_pthreads();
```

Function `request_to_use_pthreads()` must be called before the simulation starts to run for the first time, for example, before the first call of `sc_start()`. A good position in which to place the statement is at the beginning of the `sc_main()` function.

The function returns true if the request was granted. It returns false if this is not possible and also a warning is printed. Reasons may be the wrong platform (for example, linux 32-bit), or by calling the function too late.

Extensions

The following proprietary extension are available as part of VCS, and not available as part of OSCI SystemC.

- Runtime functions

Include file `systemc_user.h` contains the prototypes of functions that can be called during execution of the simulation. Add this line to your source code to make the header file visible:

```
#include <cosim/bf/systemc_user.h>
```

- `GetFullName()`

Returns the full logical name of the given object or "No Name" on error. The full name can contain hierarchical sub-paths of other domains, like Verilog:

```
namespace sc_snps {
    const char *GetFullName(sc_object *obj);
}
```

The namespace `sc_snps` is only used in SystemC version 2.2.

Note:

The corresponding member function

`sc_core::sc_object::name()` defined as part of the SystemC language usually does not return the same string as `sc_snps::GetFullName()`. Member `sc_object::name()` does not consider Verilog instances and shows only the path name w.r.t. to the SystemC hierarchy. Alternatively, the `GetFullName()` function considers the entire Verilog/SystemC instance hierarchy and gives the correct logical name of the SystemC instance inside this hierarchy.

- `GetName()`

Returns the instance name (short name) of the given object or return "No Name" on error:

```
namespace sc_snps {
    const char *GetName(sc_object *obj);
}
```

Note:

The corresponding member function

`sc_core::sc_object::basename()`, defined as part of the SystemC language, usually does not return the same string as `sc_snps::GetName()`.

- Asynchronous Reset for Clocked Thread Processes

The SystemC standard allows a clocked thread process (`SC_CTHREAD`) to have an optional synchronous reset. This is specified with the `reset_signal_is()` function as follows:

```
SC_CTHREAD( th_1, clk.pos() );
reset_signal_is( syncrst, true );
```

In addition, VCS supports an optional asynchronous reset, which is specified with the `async_reset_signal_is()` function. For example:

```
SC_CTHREAD( th_1, clk.pos() );
async_reset_signal_is( asyncrst, true );
```

Note:

This feature is a VCS-specific extension. It is not a part of the IEEE 1666 OSCI SystemC standard.

Note the following points about asynchronous resets:

- Both the synchronous and asynchronous resets are optional. A process can have one, both, or none of these resets.
- While you can specify asynchronous resets in any order, ensure that they are within the constructor section (`SCCTOR`).
- An asynchronous reset cannot be specified more than once.
- When the asynchronous reset is specified, the clocked thread process will restart if either of the following conditions are true:
 - the asynchronous reset is active during the clock edge

- the asynchronous reset changes from inactive to active even if there is no clock edge

When the synchronous reset is also specified, the process will also restart if the synchronous reset is active during the clock edge.

If only the synchronous reset is specified, the behavior is as defined in the IEEE 1666 standard.

The syntax of the asynchronous reset function is as follows:

```
async_reset_signal_is (pin, level)
```

Where:

pin

Specifies the signal, which can be either an input port or signal of type `bool`.

level

Defines the level at which the reset becomes active.

This feature has the following limitations:

- Asynchronous resets are supported only in SystemC version 2.2.
- Asynchronous reset can be specified only for a `SC_CTHREAD`.
- Asynchronous resets must be specified during elaboration, preferably within the `SCCTOR` section.

Installing VG GNU Package

VCS supports gcc compiler versions 3.3.6, 3.4.6 and 4.2.2. It supports (besides the SUN "CC" Compiler) Gnu gcc 3.3.2 on Solaris.

The FTP instructions to download VG GNU package are available in VCS/VCSMX *Release Notes* under the section *Downloading and Installing VG GNU Package* under *General Platform Support*.

Static and Dynamic Linking

The main difference between static and dynamic linking is the time at which the object files are linked into an application program. In case of static linking, object files are linked during compilation, whereas in the case of dynamic linking, linking is done at runtime.

Static Linking in VCS

You can compile C/C++/SystemC files into object files and archive them in a common object file (.a's), as shown below:

```
% g++ -O3 -Wall -I. -c ext_inv.cpp -o ext_inv.o  
% g++ -O3 -Wall -I. -c ext_buf.cpp -o ext_buf.o  
% ar -r extenv.a ext_inv.o ext_buf.o
```

Note:

Add the `-I${VCS_HOME}/include/systemc_version` option to C/C++ compiler to compile SystemC files.

The archive can be statically linked by just passing the archive as any other file on the `vcs` command line.

```
% vcs top.v ./extenv.a
```

Note:

Add the `-sysc` option to the `vcs` command line, if the object file is for SystemC.

Dynamic Linking in VCS

You can compile C/C++ files into a shared object file or you can have a pre-compiled shared object.

Note:

The pre-compiled shared object should be built on the same compiler as supported by VCS.

The shared object file uses the following naming convention:

`liblibrary_name.so.version`

It begins with the `lib` keyword, followed by any specified name `library_name`, followed by `.so.version`.

The `version` is optional and is user defined. Linker/loader automatically locates and picks the shared object file using `-L` and `-l` options as explained below.

For example, the library name in `libfoo.so` or `libfoo.so.1` is `foo`. The commands to create a shared object file are as shown:

```
% gcc -fPIC -o foo.o -c -I$VCS_HOME/include foo.c  
% gcc -shared -o libfoo.so foo.o
```

The shared object file can be dynamically linked by using `-LDFLAGS` with the `-Lpath_to_shared_object` and `-llibrary_name` options on the `vcs` command line.

`-LDFLAGS` options

Specifies the options to the linker/loader.

`-Lpath_to_shared_object`

Specifies the path, where shared objects reside.

`-llibrary_name`

Specifies the library name of the shared object file.

If there are more than one shared object located in different directories, you can specify `-Lpath_to_the_shared_object` multiple times for each directory and `-llibrary_name` multiple times for each shared object file.

```
% vcs top.v -LDFLAGS "-L<path_to_libfoo.so> -lfoo  
-L<path_to_libhello.so> -lhello"
```

You can also specify the linker options directly to the `vcs` command line.

```
% vcs top.v -Lpath_to_libfoo.so -lfoo  
-Lpath_to_libhello.so -lhello
```

Dynamic Linking in VCS

Following are the steps for dynamic linking of SystemC files:

- Create a shared object file

```
% gcc -fPIC -o foo.o -c -I$VCS_HOME/include/systemc22
      foo.cpp
% gcc -shared -o libfoo.so foo.o
```

- Analyze your SystemC top file (which is instantiated in HDL design) to create a HDL wrapper.

```
% syscan sc_top.cpp:sc_top -sysc=2.2
```

- The shared object can be dynamically linked by using the `-Lpath_to_shared_object` and `-llibrary_name` options on the `vcs` command line.

```
% vcs -sysc=2.2 top.v -Lpath_to_shared_object file -lfoo
```

LD_LIBRARY_PATH Environment Variable

You can set the `LD_LIBRARY_PATH` environment variable to the directory where the shared object file resides.

```
% setenv LD_LIBRARY_PATH
      path_to_shared_objectfile:$LD_LIBRARY_PATH
```

Now, for any change in the C/C++/SystemC files, you simply need to rebuild the shared object file with the commands as mentioned above and execute the `simv`. You do not have to rebuild the `simv`.

Limitations

The following limitations apply to the VCS/SystemC interface.

- No Donuts / Sandwiches

VCS/SystemC does not support having "donuts" or "sandwiches" in SystemC and Verilog modules. Therefore, you cannot have a SystemC instance that is instantiated under an HDL design unit and itself instantiates another HDL design unit. Similarly, a SystemC-HDL-SystemC instance hierarchy is not supported. In other words, following the design hierarchy from a leaf instance towards the root, you can transition from SystemC to HDL or vice-versa only once.

- Number of ports

There is no limitation regarding the number of port for an interface model. It may have none, one, or multiple ports.

Verilog wrapper needed for pure VHDL-top-SystemC down

The topology with VHDL-on-top and SystemC-down is supported in the UUM flow, but the following restriction is observed:

- A Verilog wrapper must be created for at least one SystemC interface model.

SystemC modules that are to be instantiated in VHDL entities are analyzed with option -vhdl in the syscan call. Example:

```
syscan -vhdl mymodel1.cpp:mymodel1
```

You can continue to use the `-vhdl` option for the majority of SystemC interface models, however, at least one module that is used within the design must be created without this option. Example:

```
syscan -vhdl mymodel1.cpp:mymodel1
syscan -vhdl mymodel2.cpp:mymodel2
syscan      mymodel3.cpp:mymodel3
vhdlan bottom.vhd top.vhd
vcs -sysc TOP
```

Note that the syscan call for mymodel3 has no "`-vhdl`" option, which means that a Verilog wrapper is created.

23

C Language Interface

It is common to mix C and C++ with both Verilog and VHDL. There are many different mechanisms and what you do will depend on your objective as well as the performance and restrictions of each mechanism. VCS supports the following ways to use C and C++ with your design:

- “Using PLI”
- “Using VPI Routines”
- “Using DirectC”
- Using SystemC - See the [Using SystemC](#) chapter.
- Using SystemVerilog DPI routines - See the [SystemVerilog LRM](#).

Note:

PLI1.0 refers to TF and ACC routines, and PLI2.0 refers to VPI.

Using PLI

PLI is the programming language interface (PLI) between C/C++ functions and VCS. It helps to link applications containing C/C++ functions with VCS, so that they execute concurrently. The C/C++ functions in the application use the PLI to read and write delay and simulation values in the VCS executable, and VCS can call these functions during simulation.

VCS supports PLI 1.0 and PLI 2.0 routines for the PLI. Therefore, you can use VPI, ACC or TF routines to write the PLI application. See Appendix E, "PLI Access Routines".

This chapter covers the following topics:

- “Writing a PLI Application”
- “Functions in a PLI Application”
- “Header Files for PLI Applications”
- “PLI Table File”
- “Enabling ACC Capabilities”

Writing a PLI Application

When writing a PLI application, you need to do the following:

1. Write the C/C++ functions of the application calling the VPI, ACC or TF routines to access data inside VCS.

2. Associate user-defined system tasks and system functions with the C/C++ functions in your application. VCS will call these functions when it compiles or executes these system tasks or system functions in the Verilog source code. In VCS, associate the user-defined system tasks and system functions with the C/C++ functions in your application using a PLI table file (see “[PLI Table File](#)” on page 6). In this file, you can also limit the scope and operations of the ACC routines for faster performance.
3. Enter the user-defined system tasks and functions in the Verilog source code.
4. Compile and simulate your design, specifying the table file and including the C/C++ source files (or compiled object files or libraries) so that the application is linked with VCS in the `simv` executable. If you include object files, use the `-cc` and `-ld` options to specify the compiler and linker that generated them. Linker errors occur if you include a C/C++ function in the PLI table file, but omit the source code for this function at compile-time.

To use the debugging features, perform the following:

1. Write a PLI table file, limiting the scope and operations of the ACC routines used by the debugging features.
2. Compile and simulate your design, specifying the table file.

These procedures are not mutually exclusive. It is, for example, quite possible that you have a PLI application that you write and use during the debugging phase of your design. If so, you can write a PLI table file that both:

- Associates user-defined system tasks or system functions with the functions in your application and limits the scope and operations called by your functions for faster performance.

- Limits scope and operations of the functions called by the debugging features in VCS.
-

Functions in a PLI Application

When you write a PLI application, you typically write a number of functions. The following are PLI functions that VCS expects with a user-defined system task or system function:

- The function that VCS calls when it executes the user-defined system task. Other functions are not necessary but this call function must be present. It is not unusual for there to be more than one call function. You'll need a separate user-defined system task for each call function. If the function returns a value then you must write a user-defined system function for it instead of a user-defined system task.
- The function that VCS calls during compilation to check if the user-defined system task has the correct syntax. You can omit this check function.
- The function that VCS calls for miscellaneous reasons such as the execution of \$stop, \$finish, or other reasons such as a value change. When VCS calls this function, it passes a reason argument to it that explains why VCS is calling it. You can omit this miscellaneous function.

These are the functions you tell VCS about in the PLI table file; apart from these PLI applications can have several more functions that are called by other functions.

Note:

You do not specify a function to determine the return value size of a user-defined system function; instead you specify the size directly in the PLI table file.

Header Files for PLI Applications

For PLI applications, you need to include one or more of the following header files:

`vpi_user.h`

For PLI Applications whose functions call IEEE Standard VPI routines as documented in the *IEEE Verilog Language Reference Manual*.

`acc_user.h`

For PLI Applications whose functions call IEEE Standard ACC routines as documented in the *IEEE Verilog Language Reference Manual*.

`vcsuser.h`

For PLI applications whose functions call IEEE Standard TF routines as documented in the *IEEE Verilog Language Reference Manual*.

`vcs_acc_user.h`

For PLI applications whose functions call the special ACC routines implemented exclusively for VCS.

These header files are located in the
`$VCS_HOME/your_platform/lib` directory.

PLI Table File

The PLI table file (also referred to as the `pli.tab` file) is used to:

- Associate user-defined system tasks and system functions with functions in a PLI application. This enables VCS to call these functions when it compiles or executes the system task or function.
- Limit the scope and operation of the PLI 1.0 or PLI 2.0 functions called by the debugging features. See “[Specifying Access Capabilities for PLI Functions](#)” on page 10 and “[Specifying Access Capabilities for VCS Debugging Features](#)” on page 15.

Syntax

The following is the syntax of the PLI table file:

```
$name PLI_specifications [access_capabilities]
```

Here:

`$name`

Specify the name of the user-defined system task or function.

`PLI_specifications`

Specify one or more specifications such as the name of the C function (mandatory), size of the return value (mandatory only for user-defined system functions), and so on. For a complete list of PLI specifications, see “[PLI Specifications](#)” on page 7.

`access_capabilities`

Specify the access capabilities of the functions defined in the PLI application. Use this to control the PLI 1.0 or PLI 2.0 functions' ability to access the design hierarchy. See [“Access Capabilities” on page 10](#) for more information.

Synopsys recommends you enable this feature while using PLIs to improve the runtime performance.

PLI Specifications

The PLI specifications are as follows:

`call=`*function*

Specifies the name of the function defined in the PLI application. This is mandatory.

`check=`*function*

Specifies the name of the check function.

`misc=`*function*

Specifies the name of the misc function.

`data=`*integer*

Specifies the value passed as the first argument to the call, check, and misc functions. The default value is 0.

Use this argument if you want more than one user-defined system task or function to use the same call, check, or misc function. In such a case, specify a different integer for each user-defined system task or function that uses the same call, check, or misc function.

`size=number`

Specifies the size of the returned value in bits. While this is mandatory for user-defined system functions, you can ignore or specify 0 for user-defined system tasks. For user-defined system functions, specify a decimal value for the number of bits. For example, `size=64`. If the user-defined system function returns a real value, specify `r`. For example, `size=r`

`args=number`

Specifies the number of arguments passed to the user-defined system task or function.

`minargs=number`

Specifies the minimum number of arguments.

`maxargs=number`

Specifies the maximum number of arguments.

`nocelldefinepli`

Disables the dumping of value change and simulation time data of modules defined under the `'celldefine` compiler directive into a VPD file created by the `$vcdpluson` system task. This capability is only used for batch simulation.

`persistent`

Checks if the specified function is defined in the PLI application, even if the corresponding system task or function is not used in the Verilog file. If the function is not found or defined in the PLI application, VCS exits with an undefined reference error message.

Note that if you use the `-debug`, `-debug_all`, or `-debug_pp` options during compilation, VCS performs these checks on every function mapped in the tab file.

To ignore this check, which is enabled by the above debug options or the `persistent` specification, set the `PERSISTENT_FLAG` environment variable to 1.

Example 1

```
$val_proc call=val_proc check=check_proc misc=misc_proc
```

In this line, VCS calls the function named `val_proc` when it executes the associated user-defined system task named `$val_proc`. It calls the `check_proc` function at compile-time to see if the user-defined system task has the correct syntax, and calls the `misc_proc` function in special circumstances like interrupts.

Example 2

```
$set_true size=16 call=set_true
```

In this line, there is an associated user-defined system function that returns a 15-bit return value. VCS calls the function named `set_true` when it executes this system function.

Note:

Do not enter blank spaces inside a PLI specification. The following copy of the last example of PLI specifications does not work:

```
$set_true size = 16 call = set_true
```

Access Capabilities

You can specify access capabilities in a PLI table file for the following reasons:

- PLI functions associated with your user-defined system task or system function. To do this, specify the access capabilities on a line in a PLI table file after the name of the user-defined system task or system function and its PLI specifications. See “[Specifying Access Capabilities for PLI Functions](#)” on page 10 for more details.
- For the debugging features VCS can use. To do this, specify access capabilities alone on a line in a PLI table file, without an associated user-defined system task or system function. See “[Specifying Access Capabilities for VCS Debugging Features](#)” on page 15 for more details.

In many ways, specifying access capabilities for your PLI functions, and specifying them for VCS debugging features, is the same. However, the capabilities that you enable, and the parts of the design to which you can apply them are different.

Specifying Access Capabilities for PLI Functions

The format for specifying access capabilities is as follows:

```
acc= | += | -= | :=capabilities:module_names [+] | %CELL | %TASK | *
```

Here:

acc

Keyword that begins a line for specifying access capabilities.

= | += | -= | :=

Operators for adding, removing, or changing access capabilities.

The operators in this syntax are as follows:

=

A shorthand for +=.

+=

Specifies adding the access capabilities that follow to the parts of the design that follow, as specified by module name, %CELL, %TASK, or * wildcard character.

- =

Specifies removing the access capabilities that follow from the parts of the design that follow, as specified by module name, %CELL, %TASK, or * wildcard character.

:=

Specifies changing the access capabilities of the parts of the design that follow, as specified by module name, %CELL, %TASK, or * wildcard character, to only those in the list of capabilities on this specification. A specification with this operator can change the capabilities specified in a previous specification.

capabilities

Comma-separated list of access capabilities. The capabilities that you can specify for the functions in your PLI specifications are as follows:

`r` or `read`

Reads the values of nets and registers in your design.

`rw` or `read_write`

Both reads from and writes to the values of registers or variables (but not nets) in your design.

`wn`

Enables writing values to nets.

`cbk` or `callback`

To be called when named objects (nets registers, ports) change value.

`cbka` or `callback_all`

To be called when named and unnamed objects (such as primitive terminals) change value.

`frc` or `force`

Forces values on nets and registers.

`prx` or `pulserx_backannotation`

Sets pulse error and pulse rejection percentages for module path delays.

`s` or `static_info`

Enables access to static information, such as instance or signal names and connectivity information. Signal values are not static information.

`tchk` or `timing_check_backannotation`

Back-annotates timing check delay values.

`gate` or `gate_backannotation`

Back-annotates delay values on gates.

`mp` or `module_path_backannotation`

Back-annotates module path delays.

`mip` or `module_input_port_backannotation`

Back-annotates delays on module input ports.

`mipb` or `module_input_port_bit_backannotation`

Back-annotates delays on individual bits of module input ports.

`module_names`

Comma-separated list of module identifiers (or names).

Specifying modules enables, disables, or changes (depending on the operator) the ability of the PLI function to use the access capability in all instances of the specified module.

+

Specifies adding, removing, or changing the access capabilities for not only the instances of the specified modules but also the instances hierarchically under the instances of the specified modules.

%CELL

Enables, disables, or changes (depending on the operator) the ability of the PLI function to use the access capability in all instances of module definitions compiled under the `celldefine compiler directive and all module definitions in Verilog library directories and library files (as specified with the -y and -v analysis options).

%TASK

Enables, disables, or changes (depending on the operator) the ability of the PLI function to use the access capability in all instances of module definitions that contain the user-defined system task or system function associated with the PLI function.

*

Enables, disables, or changes (depending on the operator) the ability of the PLI function to use the access capability throughout the entire design. Using wildcard characters could seriously impede the performance of VCS.

Note:

There are no blank spaces when specifying access capabilities.

The following examples are the PLI specification examples from the previous section with access capabilities added to them. The examples wrap to more than one line, but when you edit your PLI table file, be sure there are no line breaks in these lines.

Example 1

```
$val_proc call=val_proc check=check_proc misc=misc_proc  
acc+= rw,tchk:top,bot acc-=tchk:top
```

This example adds the access capabilities for reading and writing to nets and registers, and for back-annotating timing check delays, to these PLI functions, and enables them to do these things in all instances of modules `top` and `bot`. It then removes the access capability for back-annotating timing check delay values from these PLI functions in all instances of module `top`.

Example 2

```
$value_passer size=0 args=2 call=value_passer persistent  
acc+=rw:%TASK acc-=rw:%CELL
```

This example adds the access capability to read from and write to the values of nets and registers to these PLI functions. It enables them to do these things in all instances of modules declared in module definitions that contain the `$value_passer` user-defined system task. The example then removes the access capability to read from and write to the values of nets and registers, from these PLI functions, in module definitions compiled under the `'celldefine` compiler directive and all module definitions in Verilog library directories and library files.

Example 3

```
$set_true size=16 call=set_true acc+=rw:*
```

This example adds the access capability to read from and write to the values of nets and registers to the PLI functions. It enables them to do this throughout the entire design.

Specifying Access Capabilities for VCS Debugging Features

The format for specifying these capabilities for VCS debugging features is as follows:

```
acc= | += | -= | :=capabilities:module_names [+] | %CELL | *
```

Here:

acc

Keyword that begins a line for specifying access capabilities.

= | += | -= | :=

Operators for adding, removing, or changing access capabilities.

capabilities

Comma separated list of access capabilities.

module_names

Comma-separated list of module identifiers. The specified access capabilities will be added, removed, or changed for all instances of these modules.

+

Specifies adding, removing, or changing the access capabilities for not only the instances of the specified modules but also the instances hierarchically under the instances of the specified modules.

%CELL

Specifies all modules compiled under the `celldefine compiler directive and all modules in Verilog library directories and library files (as specified with the -y and -v options.)

*

Specifies all modules in the design. Using a wildcard character is no more efficient than using the -debug option with vcs.

The access capabilities and the interactive commands they enable are as follows:

ACC Capability	What it enables your PLI functions to do
<code>r</code> or <code>read</code>	<p>For specifying “reads” in your design, it enables commands for performing the following:</p> <ul style="list-style-type: none"> • Creating an alias for another UCLI command (<code>alias</code>) • Displaying UCLI help • Specifying the radix of displayed simulation values (<code>oformat</code>) • Displaying simulation values • Descending and ascending the module hierarchy • Depositing values on registers • Displaying the set breakpoints on signals • Displaying the port names of the current location, and the current module instance or scope, in the module hierarchy • Displaying the names of instances in the current module instance or scope • Displaying the nets and registers in the current scope • Moving up the module hierarchy • Deleting an alias for another UCLI command • Ending the simulation
<code>rw</code> or <code>read_write</code>	<p>For specifying “reads and writes” in your design but <code>r</code> enables everything that <code>rw</code> does. A longer way to specify this capability is with the <code>read_write</code> keyword.</p>

ACC Capability	What it enables your PLI functions to do
cbk or callback	<p>Commands for performing the following:</p> <ul style="list-style-type: none"> • Setting a repeating breakpoint. In other words always halting simulation, when a specified signal changes value • Setting a one shot breakpoint. In other words halting simulation the next time the signal changes value but not the subsequent times it changes value • Removing a breakpoint from a signal • Showing the line number or number in the source code of the statement or statements that causes the current value of a net • A longer way to specify this capability is with the <code>callback</code> keyword.
frc or force	<p>Commands for performing the following:</p> <ul style="list-style-type: none"> • Forcing a net or a register to a specified value so that this value cannot be changed by subsequent simulation events in the design • Releasing a net or register from its forced value <p>A longer way to specify this capability is with the <code>force</code> keyword.</p>

Example 1

The following specification enables many interactive commands including those for displaying the values of signals in specified modules and depositing values to the signals that are registers:

```
acc+=r:top,mid,bot
```

Notice that there are no blank spaces in this specification. Blank spaces cause a syntax error.

Example 2

The following specifications enable most interactive commands for most of the modules in a design. They then change the ACC capabilities preventing breakpoint and force commands in instances of modules in Verilog libraries and modules designated as cells with the `celldefine compiler directive.

```
acc+=rw, cbk, frc:top+ acc:=rw:%CELL
```

In this example, the first specification enables the interactive commands that are enabled by the `rw`, `cbk`, and `frc` capabilities for module `top`, which, in this example, is the top-level module of the design, and all module instances under it. The second specification limits the interactive commands for the specified modules to only those enabled by the `rw` (same as `r`) capability.

Using the PLI Table File

You specify the PLI table file with the `-P` compile-time option, followed by the name of the PLI table file (by convention, the PLI table file has a `.tab` extension). For example:

```
-P pli.tab
```

When you enter this option on the `vcs` command line, you can also enter C source files, compiled `.o` object files, or `.a` libraries on the `vcs` command line, to specify the PLI application that you want to link with VCS. For example:

```
vcs -P pli.tab pli.c my_design.v
```

One advantage to entering .o object files and .a libraries is that you do not have to recompile the PLI application every time you compile your design.

Enabling ACC Capabilities

As well as specifying ACC capabilities in only specific parts of your design (as described in “[PLI Table File](#)” on page 6), VCS allows you to enable ACC capabilities throughout your design. It also enables you to specify selected write capabilities using a configuration file. Since enabling ACC capabilities has an adverse effect on performance, VCS also allows you to enable only the ACC capabilities you need.

Globally

You can enter the `+acc+level_number` compile-time option to globally enable ACC capabilities throughout your design.

Note:

Using the `+acc+level_number` option significantly impedes the simulation performance of your design. Synopsys recommends that you use a PLI table file to enable ACC capabilities for only the parts of your design where you need them. For more details on doing this, see “[PLI Table File](#)” on page 6.

The `level_number` in this option specifies additional ACC capabilities as follows:

`+acc+1` or `+acc`

Enables all capabilities except value change callbacks and delay annotation.

+acc+2

Above, plus value change callbacks.

+acc+3

Above, plus module path delay annotation.

+acc+4

Above, plus gate delay annotation.

Using the Configuration File

You specify the configuration file with the `+optconfigfile` compile-time option. For example:

`+optconfigfile+filename`

The VCS configuration file enables you to enter statements that specify:

- Using the optimizations of Radiant Technology on part of a design
- Enabling PLI ACC write capabilities for all memories in the design, disabling them for the entire design, or enabling them for part or parts of the design hierarchy
- Four state simulation for part of a design

The entries in the configuration file override the ACC write-enabling entries in the PLI table file.

The syntax of each type of statement in the configuration file to enable ACC write capabilities is as follows:

```
set writeOnMem;
```

or

```
set noAccWrite;
```

or

```
module {list_of_module_identifiers} {accWrite};
```

or

```
instance {list_of_module_instance_hierarchical_names}  
{accWrite};
```

or

```
tree [(depth)] {list_of_module_identifiers} {accWrite};
```

Here:

set

Keyword preceding a property that applies to the entire design.

writeOnMem

Enables ACC write to memories (any single or multi-dimensional array of the reg data type) throughout the entire design.

noAccWrite

Disables ACC write capabilities throughout the entire design.

accWrite

Enables ACC write capabilities.

`module`

Keyword specifying that the `accWrite` attribute in this statement applies to all instances of the modules in the list, specified by module identifier.

`list_of_module_identifiers`

Comma-separated list of module identifiers (also called module names).

`instance`

Keyword specifying that the `accWrite` attribute in this statement applies to all instances in the list.

`list_of_module_instance_hierarchical_names`

Comma-separated list of module instance hierarchical names.

Note:

Follow the Verilog syntax for signal names and hierarchical names of module instances.

`tree`

Keyword specifying that the `accWrite` attribute in this statement applies to all instances of the modules in the list, specified by module identifier, and also applies to all module instances hierarchically under these module instances.

`depth`

An integer that specifies how far down the module hierarchy from the specified modules you want to apply the `accWrite` attribute. You can specify a negative value. A negative value specifies descending to the leaf level and counting up levels of the hierarchy to apply these attributes. This specification is optional. Enclose this specification in parentheses: `()`

Selected ACC Capabilities

There are compile-time and runtime options that enable VCS and PLI applications to use only the ACC capabilities they need and no more. The procedure to use these options is as follows:

1. Use the `+vcs+learn+pli` runtime option to tell VCS to keep track of, or learn, the ACC capabilities that are used by different modules in your design. VCS uses this information to create a secondary PLI table file, named `pli_learn.tab`. You can use this table file to recompile your design so that subsequent simulations use only the ACC capabilities that are needed.
2. Tell VCS to apply what it has learned in the next compilation of your design, and specify the secondary PLI table file, with the `+applylearn+filename` compile-time option (if you omit `+filename` from the `+applylearn` compile-time option, VCS uses the `pli_learn.tab` secondary PLI table file).
3. Simulate again with a `simv` executable in which only the ACC capabilities you need are enabled.

Learning What Access Capabilities are Used

You include the `+vcs+learn+pli` runtime option to tell VCS to learn the access capabilities that were used by the modules in your design and write them into a secondary PLI table file named, `pli_learn.tab`.

This file is considered a secondary PLI table file because it does not replace the first PLI table file that you used (if you used one). This file does, however, modify whatever access capabilities are specified in a first PLI table file, or other means of specifying access capabilities, so that you enable only the capabilities you need in subsequent simulations.

You should look at the contents of the `pli_learn.tab` file that VCS writes to see what access capabilities were actually used during simulation. The following is an example of this file:

```
////////////// SYNOPSYS INC ///////////
//                      PLI LEARN FILE
// AUTOMATICALLY GENERATED BY VCS (TM) LEARN MODE
///////////////
acc=r:testfixture
    //SIGNAL STIM_SRLS:r
acc=rw:SDFFR
    //SIGNAL S1:rw
```

The following line in this file specifies that during simulation, the read capability was needed for signals in the module named `testfixture`.

```
acc=r:testfixture
    //SIGNAL STIM_SRLS:r
```

The comment lets you know that the only signal for which this capability was needed was the signal named, `STIM_SRLS`. This line is in the form of a comment because the syntax of the PLI table file does not permit specifying access capabilities on a signal-by-signal basis.

The following line in this file specifies that during simulation, the read and write capabilities were needed for signals in the module named, `SDFFR`, specifically for the signal named `S1`.

```
acc=rw:SDFFR  
    //SIGNAL S1:rw
```

Signs of a Potentially Significant Performance Gain

You might see one of following comments in the `pli_learn.tab` file:

```
// !VCS_LEARNED: NO_ACCESS_PERFORMED
```

This indicates that none of the enabled access capabilities were used during the simulation.

```
// !VCS_LEARNED: NO_DYNAMIC_ACCESS_PERFORMED
```

This indicates that only static information was accessed through access capabilities and there was no value change information during simulation.

These comments indicate that there is a potentially significant performance gain when you apply the access capabilities in the `pli_learn.tab` file.

Compiling to Enable Only the Access Capabilities You Need

After you have run the simulation to learn what access capabilities were actually used by your design, you can then recompile the design with the information you have learned, so the resulting `simv` executable uses only the access capabilities you require.

When you recompile your design, include the `+applylearn` compile-time option.

If, for some reason, you renamed the `pli_learn.tab` file that VCS writes when you include the `+vcs+learn+pli` runtime option, specify the new filename in the compile-time option by appending it to the option with the following syntax:

```
+applylearn+filename
```

When you recompile your design with the `+applylearn` compile-time option, it is important that you also re-enter all the compile-time options that you used for the previous compilation. For example, if in a previous compilation, you specified a PLI table file with the `-P` compile-time option, specify this PLI table file again, using the `-P` option, along with the `+applylearn` option.

Note:

If you change your design after VCS writes the `pli_learn.tab` file, and you want to make sure that you are using only the access capabilities you need, you will need to have VCS write another one, by including the `+vcs+learn+pli` runtime option and then compiling your design again with the `+applylearn` option.

Limitations

VCS is not able maintain a history of all access capabilities. However, the capabilities it does maintain, and specify in the `pli_learned.tab` file, are as follows:

- `r` - read
- `rw` - read and write
- `cbk` - callbacks
- `cbka` - callback all including unnamed objects
- `frc` - forcing values on signals

The `+applylearn` compile-time option does not work if you also use either the `+multisource_int_delays` or `+transport_int_delays` compile-time option, because interconnect delays need global access capabilities.

If you enter the `+applylearn` compile-time option more than once on the `vcs` command line, VCS ignores all instances, except for the first occurrence.

Using VPI Routines

To enable VPI capabilities in VCS, use the compilation option `+vpi`. as shown in the following example:

```
% vcs +vpi top -P test.tab test.c
```

The header file for the VPI routines is `$VCS_HOME/include/vpi_user.h`.

You can register your user-defined system tasks/function-related callbacks using the `vpi_register_systf` VPI routine, see “[Support for the `vpi_register_systf` Routine](#)” on page 29.

You can also use a PLI .tab file to associate your user-defined system tasks with your VPI routines, see “[PLI Table File for VPI Routines](#)” on page 31.

Support for the `vpi_register_systf` Routine

VCS supports the `vpi_register_systf` VPI access routine. To use this routine, you need to make an entry in the `vpi_user.c` file. You can copy this file from `$VCS_HOME/etc/vpi`.

The following is an example::

```
/*=====
Copyright (c) 2003 Synopsys Inc
=====*/
/* Fill your start up routines in this array, Last entry
should be
zero, use -use_vpiobj to pick up this file */
extern void register_me();
void (*vlog_startup_routines[])(() = {
register_me,
    0 /* Last Entry */ };
```

entry here

In this example:

- The routine named `register_me` is externally declared.
- It is also included in the array named `vlog_startup_routines`.
- The last entry in the array is zero.

You specify this file with the `-use_vpiobj` compilation option. For example:

```
% vcs top.v -use_vpiobj vpi_user.c +vpi
```

You can also write a PLI table file for VPI routines. See “[PLI Table File for VPI Routines](#)”.

Integrating a VPI Application With VCS

If you create one or more shared libraries for a VPI application, the application should not contain the `vlog_startup_routines` array.

Instead, enter the `-load` compile-time option to specify the registration routine. The syntax is as follows:

```
-load shared_library:registration_routine
```

You do not have to specify the path name of the shared library, if that path is part of your `LD_LIBRARY_PATH` environment variable.

The following are some examples of using this option:

- `-load lib1:my_register`

The `my_register()` routine is in `lib1.so`. The location of `lib1.so` is in the `LD_LIBRARY_PATH` environment variable.

- `-load lib1:my_register,new_register`

The registration routines `my_register()` and `new_register()` are in `lib1.so`. The location of `lib1.so` is in the `LD_LIBRARY_PATH` environment variable.

- `-load lib1:my_register -load lib2:new_register`
The registration routine `my_register()` is in `lib1.so` and the second registration routine `new_register()` is in `lib2.so`. The path to both of these libraries are in the `LD_LIBRARY_PATH` environment variable. You can enter more than one `-load` option to specify multiple shared libraries and their registration routines.
- `-load lib1.so:my_register`
The registration routine `my_register()` is in `lib1.so`. The location of `lib1.so` is in the `LD_LIBRARY_PATH` environment variable.
- `-load /usr/lib/mylib.so:my_register`
The registration routine `my_register()` is in `lib1.so`, which is in `/usr/lib/mylib.so`, and not in the `LD_LIBRARY_PATH` environment variable.

PLI Table File for VPI Routines

The PLI table file for VPI routines works the same way, and with the same syntax as a PLI table file for user-defined system tasks that execute C functions. The following is an example of such a PLI table file:

```
$set_mipd_delays call=PLIbook_SetMipd_calltf
check=PLIbook_SetMipd_compiletf
acc=mip,mp,gate,tchk,rw:test+
```

Note that this entry includes `acc=` even though the C functions in the PLI specification call VPI routines instead of PLI 1.0 routines. The syntax has not changed; you use the same syntax for enabling PLI 1.0 and PLI 2.0 routines.

This PLI table file is used for an example file named `set_mipd_delays_vpi.c`, which is available with *The Verilog PLI Handbook* by Stuart Sutherland, Kluwer Academic Publishers, Boston, Dordrecht, and London.

Unimplemented VPI Routines

VCS has not implemented everything specified for VPI routines in the *IEEE Verilog Language Reference Manual*, because some routines would be rarely used and some of the data access operations of other routines would be rarely used. The unimplemented routines are as follows:

- `vpi_get_data`
- `vpi_put_data`
- `vpi_sim_control`

Object data model diagrams in the *IEEE Verilog Language Reference Manual* specify that some VPI routines should be able to access data that is rarely needed. These routines, and the data they cannot access, are as follows:

`vpi_get_value`

- Cannot retrieve the value of `var` select objects (diagram 26.6.8 Variables) and `func` call objects (diagram 26.6.18 Task, function declaration).
- Cannot retrieve the value of VPI operators (expressions) unless they are arguments to system tasks or system functions.
- Cannot retrieve the value of UDP table entries (`vpiVectorVal` not implemented).

vpi_put_value

Cannot set the value of var select objects (diagram 26.6.8 Variables) and primitive objects (diagram 26.6.13 Primitive, prim term).

vpi_get_delays

Cannot retrieve the values of continuous assign objects (diagram 26.6.24 Continuous assignment) or procedurally assigned objects.

vpi_put_delays

Cannot put values on continuous assign objects (diagram 26.6.24 Continuous assignment) or procedurally assigned objects.

vpi_register_cb

Cannot register the following types of callbacks that are defined for this routine:

cbEndOfSimulation	cbError	cbPliError
cbTchkViolation	cbSignal	cbForce
cbRelease	cbAssign	cbDeassign

Also, the cbValueChange callback is not supported for the following objects:

- A memory or a memory word (index or element)
- VarArray or VarSelect

Using DirectC

DirectC is an extended interface between Verilog and C/C++. It is an alternative to the PLI that, unlike the PLI, enables you to do the following:

- More efficiently pass values between Verilog module instances and C/C++ functions by calling the functions directly, along with actual parameters, in your Verilog code.
- Pass more types of data between Verilog and C/C++. With the PLI, you can only pass Verilog information to and from a C/C++ application. With DirectC you do not have this limitation.

With DirectC, for example, you can model a simulation environment for your design in C/C++ in which you can pass pointers from the environment to your design and store them in Verilog signals, and at a later simulation time, pass these pointers to the simulation environment.

Similarly, you can use DirectC to develop applications to run with VCS to which you can pass pointers to the location of simulation values for your design.

DirectC is an alternative to, but not a replacement for, the PLI. You can do things with the PLI that you cannot do with DirectC. For example, there are PLI TF and ACC routines to implement a callback to start a C/C++ function when a Verilog signal changes value. You cannot do this with DirectC.

You can use Direct C/C++ function calls for existing and proven C code as well as C/C++ code that you write in the future. You can also use them without much rewriting of, or additions to, your Verilog code. You call them the same way you call (or enable) a Verilog function or Verilog task.

This section describes the DirectC interface in the following sections:

- “Using Direct C/C++ Function Calls”
- “Using Direct Access”
- “Using Abstract Access”
- “Enabling C/C++ Functions”
- “Extended BNF for External Function Declarations”

Using Direct C/C++ Function Calls

To enable a direct call of a C/C++ function during simulation, perform the following:

1. Declare the function in your Verilog code.
2. Call the function in your Verilog code.
3. Compile your design and C/C++ code using compile-time options for DirectC.

However, there are complications to this otherwise straightforward procedure.

DirectC allows the invocation of C++ functions that are declared in C++ using the `extern "C"` linkage directive. The `extern "C"` directive is necessary to protect the name of the C++ function from being mangled by the C++ compiler. Plain C functions do not undergo mangling, and therefore, do not need any special directive.

The declaration of these functions involves specifying a direction for the parameters of the C function, because, in the Verilog environment, they become analogous to Verilog tasks as well as functions. Verilog tasks are similar to void C functions in that they do not return a value. However, Verilog tasks do have input, output, and inout arguments, whereas C function parameters do not have explicitly declared directions. See “[Declaring the C/C++ Function](#)”.

There are two access modes for C/C++ function calls. These modes do not make much difference in your Verilog code; they only pertain to the development of the C/C++ function. They are as follows:

- The slightly more efficient direct access mode - this mode has rules for how values of different types and sizes are passed to and from Verilog and C/C++. This mode is explained in detail in the section, “[Using Direct Access](#)”.
- The slightly less efficient, but with better error handling abstract access mode - in this implementation, VCS creates a descriptor for each actual parameter of the C function. You access these descriptors using a specially defined pointer called a handle. All formal arguments are handles. DirectC comes with a library of accessory functions for using these handles. This mode is explained in detail in the section, “[Using Abstract Access](#)”.

The abstract access library of accessory functions contains operations for reading and writing values and for querying about argument types, sizes, etc. An alternative library, with perhaps different levels of security or efficiency, can be developed and used in abstract access without changing your Verilog or C/C++ code.

If you have an existing C/C++ function that you want to use in a Verilog design, you consider using direct access and see if you really need to edit your C/C++ function or write a wrapper so that you can use direct access inside the wrapper. There is a small performance gain by using direct access compared to abstract access.

If you are about to write a C/C++ function to use in a Verilog design, first decide how you wish to use it in your Verilog code and write the external declaration for it, then decide which access mode you want. You can change the mode later with perhaps a small change in your Verilog code.

Using abstract access is “safer” because the library of accessory functions for abstract access has error messages to help you to debug the interface between C/C++ and Verilog. With direct access, errors simply result in segmentation faults, memory corruption, etc.

Abstract access can be generalized more easily for your C/C++ function. For example, with open arrays you can call the function with 8-bit arguments at one point in your Verilog design and call it again some place else with 32-bit arguments. The accessory functions can manage the differences in size. With abstract access you can have the size of a parameter returned to you. With direct access you must know the size.

How C/C++ Functions Work in a Verilog Environment

Like Verilog functions, and unlike Verilog tasks, no simulation time elapses during the execution of a C/C++ function.

C/C++ functions work in two-state and four-state simulation, and in some cases, work better in two-state simulation. Short vector values, 32-bits or less, are passed by value instead of by reference. Using two-state simulation makes a difference in how you declare a C/C++ function in your Verilog code.

The parameters of C/C++ functions, are analogous to the arguments of Verilog tasks. They can be input, output, or inout just like the arguments of Verilog tasks. You don't specify them as such in your C code, but you do when you declare them in your Verilog code. Accordingly your Verilog code can pass values to parameters declared to be input or inout, but not output, in the function declaration in your Verilog code, and your C function can only pass values from parameters declared to be inout or output, but not input, in the function declaration in your Verilog code.

If a C/C++ function returns a value to a Verilog register (the C/C++ function is in an expression that is assigned to the register) the return value of the C/C++ function is restricted to the following:

- The value of a scalar `reg` or `bit`

Note:

In two-state simulation, a `reg` has a new name, `bit`.

- The value of the C type `int`
- A pointer
- A short, 32 bits or less, vector `bit`

- The value of a Verilog `real` which is represented by the C type `double`

So C/C++ functions cannot return the value of a four-state vector `reg`, long (longer than 32 bits) vector `bit`, or Verilog `integer`, `realtime`, or `time` data type. You can pass these type of values out of the C/C++ function using a parameter that you declare to be inout or output in the declaration of the function in your Verilog code.

Declaring the C/C++ Function

A partial EBNF specification for external function declaration is as follows:

```
source_text      ::= description +
description       ::= module | user_defined_primitive | extern_declaration
extern_declaration ::= extern access_mode ? attribute ? return_type function_id
                      (extern_func_args ? ) ;
access_mode       ::= ( "A" | "C" )
attribute         ::= pure
return_type        ::= void | reg | bit | DirectC_primitive_type
                      | small_bit_vector
small_bit_vector  ::= bit [ (constant_expression : constant_expression) ]
extern_func_args  ::= extern_func_arg ( , extern_func_arg ) *
extern_func_arg   ::= arg_direction ? arg_type arg_id ?
arg_direction     ::= input | output | inout
arg_type          ::= bit_or_reg_type | array_type | DirectC_primitive_type
bit_or_reg_type   ::= ( bit | reg ) optional_vector_range ?
optional_vector_range ::= [ ( constant_expression : constant_expression ) ? ]
array_type         ::= bit_or_reg_type array [ (constant_expression :
                      constant_expression) ? ]
DirectC_primitive_type ::= int | real | pointer | string
```

Here:

extern

Keyword that begins the declaration of the C/C++ function declaration.

access_mode

Specifies the mode of access in the declaration. Enter C for direct access, or A for abstract access. Using this entry enables some functions to use direct access and others to use abstract access.

attribute

An optional attribute for the function. The `pure` attribute enables some optimizations. Enter this attribute if the function has no side effects and is dependent only on the values of its input parameters.

return_type

The valid return types are `int`, `bit`, `reg`, `string`, `pointer`, and `void`. See [Table 23-1](#) for a description of what these types specify.

small_bit_vector

Specifies a bit-width of a returned vector `bit`. A C/C++ function cannot return a four-state vector `reg`, but it can return a vector `bit` if its bit-width is 32 bits or less.

function_id

The name of the C/C++ function.

direction

One of the following keywords: `input`, `output`, `inout`. In a C/C++ function, these keywords specify the same thing that they specify in a Verilog task; see [Table 23-2](#).

arg_type

The valid argument types are `real`, `reg`, `bit`, `int`, `pointer`, `string`.

[bit_width]

Specifies the bit-width of a vector `reg` or `bit` that is an argument to the C/C++ function. You can leave the bit-width open by entering `[]`.

array

Specifies that the argument is a Verilog memory.

[index_range]

Specifies a range of elements (words, addresses) in the memory. You can leave the range open by entering `[]`.

arg_id

The Verilog register argument to the C/C++ function that becomes the actual parameter to the function.

Note:

Argument direction (i.e., `input`, `output`, `inout`) applies to all arguments that follow it until the next direction occurs; the default direction is `input`.

Table 23-1 C/C++ Function Return Types

Return Type	Specifies
<code>int</code>	The C/C++ function returns a value for type <code>int</code> .
<code>bit</code>	The C/C++ function returns the value of a bit, which is a Verilog <code>reg</code> in two state simulation, if it is 32 bits or less.
<code>reg</code>	The C/C++ function returns the value of a Verilog scalar <code>reg</code> .
<code>string</code>	The C/C++ function returns a pointer to a character string.
<code>pointer</code>	The C/C++ function returns a pointer.
<code>void</code>	The C/C++ function does not return a value.

Table 23-2 C/C++ Function Argument Directions

keyword	Specifies
input	The C/C++ function can only read the value or address of the argument. If you specify an input argument first, you can omit the <code>input</code> keyword.
output	The C/C++ function can only write the value or address of the argument.
inout	The C/C++ function can both read and write the value or address of the argument.

Table 23-3 C/C++ Function Argument Types

keyword	Specifies
real	The C/C++ function reads or writes the address of a Verilog real data type.
reg	The C/C++ function reads or writes the value or address of a Verilog reg.
bit	The C/C++ function reads or writes the value or address of a Verilog reg in two state simulation.
int	The C/C++ function reads or writes the address of a C/C++ int data type.
pointer	The C/C++ function reads or writes the address that a pointer is pointing to.
string	The C/C++ function reads from or writes to the address of a string.

Example 1

```
extern "A" reg return_reg (input reg r1);
```

This example declares a C/C++ function named `return_reg`. This function returns the value of a scalar reg. When we call this function, the value of a scalar reg named `r1` is passed to the function. This function uses abstract access.

Example 2

```
extern "C" bit [7:0] return_vector_bit (bit [7:0] r3);
```

This example declares a C/C++ function named `return_vector_bit`. This function returns an 8-bit vector bit (a reg in two state simulation). When we call this function, the value of an 8-bit vector bit (a reg in two state simulation) named `r3` is passed to the function. This function uses direct access.

The keyword `input` is omitted. This keyword can be omitted if the first argument specified is an input argument.

Example 3

```
extern string return_string();
```

This example declares a C/C++ function named `return_string`. This function returns a character string and takes no arguments.

Example 4

```
extern void receive_string( input string r5);
```

This example declares a C/C++ function named `receive_string`. It is a void function. At some time earlier in the simulation, another C/C++ function passed the address of a character string to reg `r5`. When we call this function, it reads the address in reg `r5`.

Example 5

```
extern pointer return_pointer();
```

This example declares a C/C++ function named `return_pointer`. When we call this function, it returns a pointer.

Example 6

```
extern void receive_pointer (input pointer r6);
```

This example declares a C/C++ function named `receive_pointer`. When we call this function the address in reg `r6` is passed to the function.

Example 7

```
extern void memory_reorg (input bit [32:0] array [7:0] mem2,  
output bit [32:0] array [7:0] mem1);
```

This example declares a C/C++ function named `memory_reorg`. When we call this function, the values in memory `mem2` are passed to the function. After the function executes, new values are passed to memory `mem1`.

Example 8

```
extern void incr (inout bit [] r7);
```

This example declares a C/C++ function named `incr`. When we call this function, the value in bit `r7` is passed to the function. When it finishes executing, it passes a new value to bit `r7`. We did not specify a bit width for vector bit `r7`. This allows us to use various sizes in the parameter declaration in the C/C++ function header.

Example 9

```
extern void passbig (input bit [63:0] r8,  
output bit [63:0] r9);
```

This example declares a C/C++ function named `passbig`. When we call this function, the value in bit `r8` is passed by reference to the function because it is more than 32 bits; see “[Using Direct Access on page 54](#)”. When it finishes executing, a new value is passed by reference to bit `r9`.

Calling the C/C++ Function

After declaring the C/C++ function, you can call it in your Verilog code. You call a void C/C++ function in the same manner as you call a Verilog task-enabling statement, that is, by entering the function name and its arguments, either on a separate line in an `always` or `initial` block, or in the procedural statements in a Verilog task or function declaration. Unlike Verilog tasks, you can call a C/C++ function in a Verilog function.

You call a non-void (returns a value) C/C++ function in the same manner as you call a Verilog function call, that is, by entering its name and arguments, either in an expression on the RHS of a procedural assignment statement in an `always` or `initial` block, or in a Verilog task or function declaration.

Examples

```
r2=return_reg(r1);
```

The value of scalar reg `r1` is passed to C/C++ function `return_reg`. It returns a value to reg `r2`.

```
r4=return_vector_bit(r3);
```

The value of vector bit `r3` is passed to C/C++ function `return_vector_bit`. It returns a value to vector bit `r4`.

```
r5=return_string();
```

The address of a character string is passed to reg `r5`.

```
receive_string(r5);
```

The address of a character string in reg `r5` is passed to C/C++ function `receive_string`.

```
r6=return_pointer();
```

The address pointed to in a pointer in C/C++ function `return_pointer` is passed to reg `r6`.

```
get_pointer(r6);
```

The address in reg `r6` is passed to C/C++ function `get_pointer`.

```
memory_reorg(mem1,mem2);
```

In this example, all the values in memory `mem2` are passed to C/C++ function `memory_reorg`, and when it finishes executing, it passes new values to memory `mem1`.

```
incr(r7);
```

In this example, the value of bit `r7` is passed to C/C++ function `incr`, and when it finishes executing, it passes a new value to bit `r7`.

Storing Vector Values in Machine Memory

Users of direct access need to know how vector values are stored in memory. This information is also helpful for users of abstract access.

Verilog four-state simulation values (1, 0, x, and z) are represented in machine memory with data and control bits. The control bit differentiates between the 1 and x and the 0 and z values, as shown in the following table:

Simulation Value	Data Bit	Control Bit
1	1	0
x	1	1
0	0	0
z	0	1

When a routine returns Verilog data to a C/C++ function, how that data is stored depends on whether it is from a two-state or four-state value, and whether it is from a scalar, a vector, or from an element in a Verilog memory.

For a four-state vector (denoted by the keyword reg), the Verilog data is stored in type `vec32`, which for abstract access is defined as follows:

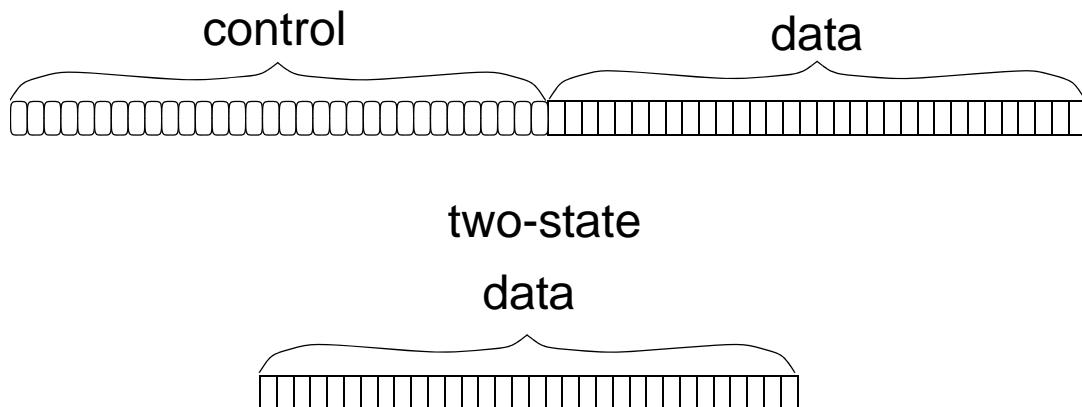
```
typedef unsigned int U;
typedef struct { U c; U d; } vec32;
```

So, type `vec32*` has two members of type `U`; member `c` is for control bits and member `d` is for data bits.

For a two-state vector bit, the Verilog data is stored in type `U*`.

Vector values are stored in arrays of chunks of 32 bits. For four-state vectors there are chunks of 32 bits for data values and 32 bits for control values. For two-state vectors, there are chunks of 32 bits for data values.

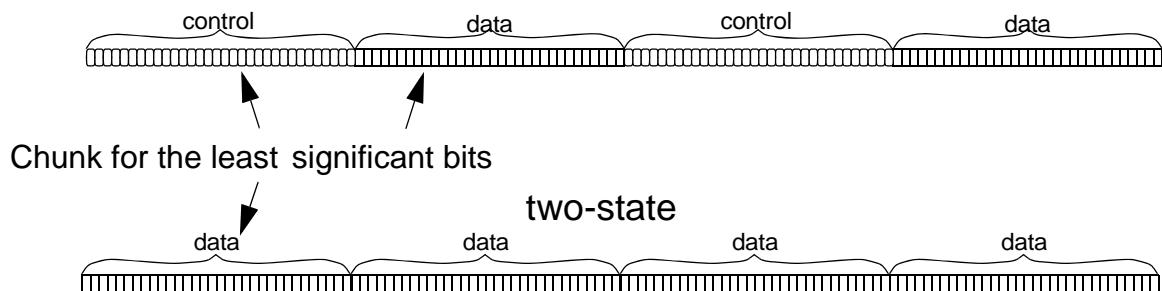
Figure 23-1 Storing Vector Values
four-state



Long vectors, more than 32 bits, have their value stored in more than one group of 32 bits and can be accessed by chunk. Short vectors, 32 bits or less, are stored in a single chunk.

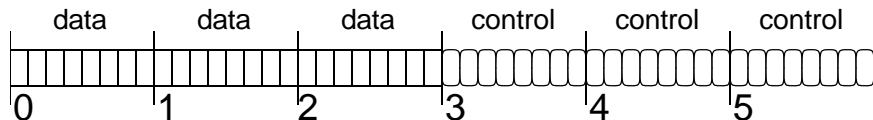
For long vectors, the chunk for the least significant bits come first, followed by the chunks for the more significant bits.

*Figure 23-2 Storing Vector Values of More than 32 Bits
four-state*



In an element in a Verilog memory, for each eight bits in the element, there is a data byte and a control byte with an additional set of bytes for remainder bit. So, if a memory had 9 bits, it would need two data bytes and two control bytes. If it had 17 bits, it would need three data bytes and three control bytes. All the data bytes precede the control bytes. Two-state memories have both data and control bytes, but the bits in the control bytes always have a zero value.

Figure 23-3 Storing Verilog Memory Elements in Machine Memory



Converting Strings

There are no *true* strings in Verilog, and a string literal, like "some_text," is just a notation for vectors of bits, based on the same principle as binary, octal, decimal, and hexadecimal numbers. So there is a need for a conversion between the two representations of "strings": the C-style representation (which actually is a pointer to the sequence of bytes terminated with null byte) and the Verilog vector encoding a string.

DirectC comes with the `vc_ConvertToString()` routine that you can use to convert a Verilog string to a C string. Its syntax is as follows:

```
void vc_ConvertToString(vec32 *, int, char *)
```

There are scenarios in which a string is created on the Verilog side and is passed to C code, and therefore, has to be converted from Verilog representation to C representation. Consider the following example:

```
extern void WriteReport(string result_code, .... /* other stuff */);
```

Example of a valid call:

```
WriteReport("Passes", ....);
```

Example of incorrect code:

```
reg [100*8:1] message;  
.  
.  
.message = "Failed";  
.  
.  
.WriteReport(message, ....);
```

This call causes a core dump because the function expects a pointer and gets some random bits instead.

It may happen that a string, or different strings, are assigned to a signal in Verilog code and their values are passed to C. For example:

```
task DoStuff(..., result_code); ... output reg [100*8:1]
```

```

result_code;
begin
.
.
.
if (...) result_code = "Bus error";
.
.
.
if (...) result_code = "Erroneous address";
.
.
.
else result_code = "Completed");
end
endtask

reg [100*8:1] message;

.....
DoStuff(..., message);

```

You cannot directly call the function as follows:

```
WriteReport(message, ...)
```

There are two solutions:

Solution 1: Write a C wrapper function, pass "message" to this function and perform the conversion of vector-to-C string in C, calling vc_ConvertToString.

Solution 2: Perform the conversion on the Verilog side. This requires some additional effort, as the memory space for a C string has to be allocated as follows:

```
extern "C" string malloc(int);
extern "C" void vc_ConvertToString(reg [], int, string);
// this function comes from DirectC library

reg [31:0] sptr;
.

.

.

// allocate memory for a C-string
sptr = malloc(8*100+1);
//100 is the width of 'message', +1 is for NULL terminator
// perform conversion
vc_ConvertToString(message, 800, sptr);
WriteReport(sptr, ...);
```

Avoiding a Naming Problem

In a module definition, do not call an external C/C++ function with the same name as the module definition. The following is an example of the type of source code you should avoid:

```
extern void receive_string (input string r5);
.
.
.
module receive_string;
.
.
.
always @ r5
begin
.
.
.
receive_string(r5) ;
.
.
.
end
endmodule
```

Using Direct Access

Direct access was implemented for C/C++ routines whose formal parameters are of the following types:

int	int*	double*	void*	void**
char*	char**	scalar	scalar*	
U*	vec32	UB*		

Some of these type identifiers are standard C/C++ types; those that are not, were defined with the following `typedef` statements:

```
typedef unsigned int U;
typedef unsigned char UB;
typedef unsigned char scalar;
typedef struct {U c; U d;} vec32;
```

The type identifier you use depends on the corresponding argument direction, type, and bit-width that you specified in the declaration of the function in your Verilog code. The following rules apply:

- Direct access passes all output and inout arguments by reference, so their corresponding formal parameters in the C/C++ function must be pointers.
- Direct access passes a Verilog bit by value only if it is 32 bits or less. If it is larger than 32 bits, direct access passes the bit by reference so the corresponding formal parameters in the C/C++ function must be pointers if they are larger than 32 bits.
- Direct access passes a scalar reg by value. It passes a vector reg direct access by reference, so the corresponding formal parameter in the C/C++ function for a vector reg must be a pointer.
- An open bit-width for a reg makes it possible for you to pass a vector reg, so the corresponding formal parameter for a reg argument, specified with an open bit-width, must be a pointer. Similarly, an open bit-width for a bit makes it possible for you to pass a bit larger than 32 bits, so the corresponding formal parameter for a bit argument specified with an open bit width must be a pointer.
- Direct access passes by value the following types of input arguments: `int`, `string`, and `pointer`.
- Direct access passes input arguments of type `real` by reference.

The following tables show the mapping between the data types you use in the C/C++ function and the arguments you specify in the function declaration in your Verilog code.

Table 23-4 For Input Arguments

argument type	C/C++ formal parameter data type	Passed by
int	int	value
real	double*	reference
pointer	void*	value
string	char*	value
bit	scalar	value
reg	scalar	value
bit [] - 1-32 bit wide vector	U	value
bit [] - open vector, any vector wider than 32 bits	U*	reference
reg [] - 1-32 bit wide vector	vec32*	reference
array [] - open vector, any vector wider than 32 bits	UB*	reference

Table 23-5 For Output and Inout Arguments

argument type	C/C++ formal parameter data type	Passed by
int	int*	reference
real	double*	reference
pointer	void**	reference
string	char**	reference
bit	scalar*	reference
reg	scalar*	reference
bit [] - any vector, including open vector	U*	reference

Table 23-5 For Output and Inout Arguments

argument type	C/C++ formal parameter data type	Passed by
reg [] - any vector, including open vector	vec32*	reference
array [] - any array, 2 state or 4 state, including open array	UB*	reference

In direct access, the return value of the function is always passed by value. The data type of the returned value is the same as the input argument.

Example 1

Consider the following C/C++ function declared in the Verilog source code:

```
extern reg return_reg (input reg r1);
```

In this example, the function named `return_reg` returns the value of a scalar reg. The value of a scalar reg is passed to it. The header of the C/C++ function is as follows:

```
extern "C" scalar return_reg(scalar reti);
scalar return_reg(scalar reti);
```

If `return_reg()` is a C++ function, it must be protected from name mangling, as follows:

```
extern "C" scalar return_reg(scalar reti);
```

Note:

The `extern "C"` directive has been omitted in subsequent examples, for brevity.

A scalar reg is passed by value to the function so the parameter is not a pointer. The parameter's type is scalar.

Example 2

Consider the following C/C++ function declared in the Verilog source code:

```
extern "C" bit [7:0] return_vector_bit (bit [7:0] r3);
```

In this example, the function named `return_vector_bit` returns the value of a vector bit. The "C" entry specifies direct access.

Typically, a declaration includes this when some other functions use abstract access. The value of an 8-bit vector bit is passed to it. The header of the C/C++ function is as follows:

```
U return_vector_bit(U returner);
```

A vector bit is passed by value to the function because the vector bit is less than 33 bits so the parameter is not a pointer. The parameter's type is U.

Example 3

Consider the following C/C++ function declared in the Verilog source code:

```
extern void receive_pointer ( input pointer r6 );
```

In this example, the function named `receive_pointer` does not return a value. The argument passed to it is declared to be a pointer. The header of the C/C++ function is as follows:

```
void receive_pointer(*pointer_receiver);
```

A pointer is passed by value to the function so the parameter is a pointer of type `void`, a generic pointer. In this example, we don't need to know the type of data that it points to.

Example 4

Consider the following C/C++ function declared in the Verilog source code:

```
extern void memory_rewriter (input bit [1:0] array [1:0]
                             mem2, output bit [1:0] array [1:0] mem1);
```

In this example, the function named `memory_rewriter` has two arguments, one declared as an input, the other as an output. Both arguments are bit memories. The header of the C/C++ function is as follows:

```
void memory_rewriter(UB *out[2], *in[2]);
```

Memories are always passed by reference to a C/C++ function so the parameter named `in` is a pointer of type `UB` with the size that matched the memory range. The parameter named `out` is also a pointer, because its corresponding argument is declared to be output. Its type is also `UB` because it outputs to a Verilog memory.

Example 5

Consider the following C/C++ function declared in the Verilog source code:

```
extern void incr (inout bit [] r7);
```

In this example, the function named `incr`, that does not return a value, has an argument declared as `inout`. No bit-width is specified, but the `[]` entry for the argument specifies that it is not a scalar bit. The header of the C/C++ function is as follows:

```
void incr (U *p);
```

Open bit-width parameters are always passed to by reference. A parameter whose corresponding argument is declared to be `inout` is passed to and from by reference. So there are two reasons for parameter `p` to be a pointer. It is a pointer to type `U` because its corresponding argument is a vector bit.

Example 6

Consider the following C/C++ function declared in the Verilog source code:

```
extern void passbig1 (input bit [63:0] r8,  
                      output bit [63:0] r9);
```

In this example, the function named `passbig1`, that does not return a value, has input and output arguments declared as `bit` and larger than 32 bits. The header of the C/C++ function is as follows:

```
void passbig (U *in, U *out)
```

In this example, the parameters `in` and `out` are pointers to type `U`. They are pointers because their corresponding arguments are larger than 32 bits and type `U` because their corresponding arguments are type `bit`.

Example 7

Consider the following C/C++ function declared in the Verilog source code:

```
extern void passbig2 (input reg [63:0] r10,  
                      output reg [63:0] r11);
```

In this example, the function named `passbig2`, that does not return a value, has input and output arguments declared as non-scalar `reg`. The header of the C/C++ function is as follows:

```
void passbig2(vec32 *in, vec32 *out)
```

In this example, the parameters `in` and `out` are pointers to type `vec32`. They are pointers because their corresponding arguments are non-scalar type `reg`.

Example 8

Consider the following C/C++ function declared in the Verilog source code:

```
extern void reality (input real real1, output real real2);
```

In this example, the function named `reality`, that does not return a value, has `input` and `output` arguments of declared type `real`. The header of the C/C++ function is as follows:

```
void reality (double *in, double *out)
```

In this example, the parameters `in` and `out` are pointers to type `double` because their corresponding arguments are type `real`.

Using the vc_hdrs.h File

When you compile your design for DirectC (by including the `+vc` compile-time option), VCS writes a file in the current directory named `vc_hdrs.h`. In this file, there are `extern` declarations for all the C/C++ functions that you declared in your Verilog code. For example, if you compile the Verilog code that contains all the C/C++ declarations in the examples in this section, the `vc_hdrs.h` file contains the following `extern` declarations:

```
extern void memory_rewriter(UB* mem2, /*OUT*/UB* mem1);  
extern U return_vector_bit(U r3);  
extern void receive_pointer(void* r6);  
extern void incr(/*INOUT*/U* r7);  
extern void* return_pointer();  
extern scalar return_reg(scalar r1);  
extern void reality(double* real1, /*OUT*/double* real2);  
extern void receive_string(char* r5);  
extern void passbig2(vec32* r8, /*OUT*/vec32* r9);  
extern char* return_string();  
extern void passbig1(U* r8, /*OUT*/U* r9);
```

These declarations contain the `/*OUT*/` comment in the parameter specification if its corresponding argument in your Verilog code is of type `output` in the declaration of the function.

These declarations contain the `/*INOUT*/` comment in the parameter specification if its corresponding argument in your Verilog code is of type `inout` in the declaration of the function.

You can copy from these `extern` declarations to the function headers in your C code. If you do, you will always use the right type of parameter in your function header and you do not have to learn the rules for direct access. Let VCS do this for you.

Access Routines for Multi-Dimensional Arrays

DirectC requires that Verilog multi-dimensional arrays be linearized (turned into arrays of the same size, but with only one dimension). VCS provides routines for obtaining information about Verilog multi-dimensional arrays when using direct access. This section describes these routines.

UB *vc_arrayElemRef(UB*, U, ...)

The UB* parameter points to an array, either a single dimensional array or a multi-dimensional array, and the U parameters specify indices in the multi-dimensional array. This routine returns a pointer to an element of the array or NULL if the indices are outside the range of the array or there is a null pointer.

```
U dgetelem(UB *mem_ptr, int i, int j) {
    int idx;
    U k;
    /* remaining indices are constant */
    UB *p = vc_arrayElemRef(mem_ptr,i,j,0,1);
    k = *p;
    return(k);
}
```

There are specialized versions of this routine for one-, two-, and three-dimensional arrays:

```
UB *vc_array1ElemRef(UB*, U)
UB *vc_array2ElemRef(UB*, U, U)
UB *vc_array3ElemRef(UB*, U, U, U)
```

U vc_getSize(UB*,U)

This routine is similar to the vc_mdaSize() routine used in abstract access. It returns the following:

- If the U type parameter has a value of 0, it returns the number of indices in an array.
- If the U type parameter has a value greater than 0, it returns the number of values in the index specified by the parameter. There is an error condition if this parameter is out of the range of indices.

If the UB pointer is null, this routine returns 0.

Using Abstract Access

In abstract access, VCS creates a descriptor for each argument in a function call. The corresponding formal parameters in the function uses a specially defined pointer to these descriptors called `vc_handle`. In abstract access, you use these “handles” to pass data and values by reference to and from these descriptors.

The idea behind abstract access is that you do not have to worry about the type you use for parameters, because you always use a special pointer type called `vc_handle`.

In abstract access, VCS creates a descriptor for every argument that you enter in the function call in your Verilog code. The `vc_handle` is a pointer to the descriptor for the argument. It is defined as follows:

```
typedef struct VeriC_Descriptor *vc_handle;
```

Using `vc_handle`

In the function header, the `vc_handle` for a Verilog reg, bit, or memory is based on the order that you declare the `vc_handle` and the order that you entered its corresponding reg, bit, or memory in the function call in your Verilog code. For example, you could have declared the function and called it in your Verilog code as follows:

```

extern "A" void my_function( input bit [31:0] r1,
                            input bit [32:0] r2);

module dev1;
reg [31:0] bit1;
reg [32:0] bit2;
initial
begin
.
.
.
my_function(bit1,bit2);
.
.
.
end
endmodule

```

Declare the function

Enter first bit1 then bit2 as arguments in the function call

This is using abstract access so VCS created descriptors for bit1 and bit2. These descriptors contain information about their value, but also other information such as whether they are scalar or vector, and whether they are simulating in two- or four-state simulation.

The corresponding header for the C/C++ function is as follows:

```

.
.
.
my_function(vc_handle h1, vc_handle h2)
{
.
.
.
    up1=vc_2stVectorRef(h1);
    up2=vc_2stVectorRef(h2);
.
.
.
}

```

**h1 is the vc_handle for bit1
h2 is the vc_handle for bit2**

A routine that accesses the data structures for bit1 and bit2 using their vc_handles

After declaring the vc_handles, you can use them to pass data to and from these descriptors.

Using Access Routines

Abstract access comes with a set of access routines that enable your C/C++ function to pass values to and from the descriptors for the Verilog reg, bit, and memory arguments in the function call.

These access routines use the vc_handle to pass values by reference, but the vc_handle is not the only type of parameter for many of these routines. These routines also have the following types of parameters:

- Scalar — an unsigned char
- Integers — uninterpreted 32 bits with no implied semantics
- Other types of pointers — primitive types “string” and “pointer”
- Real numbers

The access routines were named to help you to remember their function. Routine names beginning with vc_get are for retrieving data from the descriptor for the Verilog parameter. Routine names beginning with vc_put are for passing new values to these descriptors.

These routines can convert Verilog representation of simulation values and strings to string representation in C/C++. Strings can also be created in a C/C++ function and passed to Verilog, but you should keep in mind that they can be overwritten in Verilog. Therefore, you should copy them to local buffers if you want them to persist.

The following are the access routines, their parameters, and return values, and examples of how they are used. There is a summary of the access routines at the end of this chapter; see “[Summary of Access Routines](#)”.

int vc_isScalar(vc_handle)

Returns a 1 value if the `vc_handle` is for a one-bit reg or bit; returns a 0 value for a vector reg or bit or any memory including memories with scalar elements. For example:

```
extern "A" void scalarfinder(input reg r1,
                               input reg [1:0] r2,
                               input reg [1:0] array [1:0] r3,
                               input reg array [1:0] r4);

module top;
reg r1;
reg [1:0] r2;
reg [1:0] r3 [1:0];
reg r4 [1:0];
initial
scalarfinder(r1,r2,r3,r4);
endmodule
```

In this example, we declare a routine named `scalarfinder` and input a scalar reg, a vector reg and two memories (one with scalar elements).

The declaration contains the "A" specification for abstract access. You typically include it in the declaration when other functions will use direct access, that is, you have a mix of functions with direct and abstract access.

```
#include <stdio.h>
#include "DirectC.h"

scalarfinder(vc_handle h1, vc_handle h2, vc_handle h3,
```

```

vc_handle h4)
{
int i1 = vc_isScalar(h1),
    i2 = vc_isScalar(h2),
    i3 = vc_isScalar(h3),
    i4 = vc_isScalar(h4);
printf("\ni1=%d i2=%d i3=%d i4=%d\n\n",i1,i2,i3,i4);
}

```

Parameters `h1`, `h2`, `h3`, and `h4` are `vc_handles` to regs `r1` and `r2` and memories `r3` and `r4`, respectively. The function prints the following:

```
i1=1 i2=0 i3=0 i4=0
```

int vc_isVector(vc_handle)

This routine returns a 1 value if the `vc_handle` is to a vector reg or bit. It returns a 0 value for a vector bit or reg or any memory. For example, using the Verilog code from the previous example, and the following C/C++ function:

```

scalarfinder(vc_handle h1, vc_handle h2, vc_handle h3,
vc_handle h4)
{
int i1 = vc_isVector(h1),
    i2 = vc_isVector(h2),
    i3 = vc_isVector(h3),
    i4 = vc_isVector(h4);
printf("\ni1=%d i2=%d i3=%d i4=%d\n\n",i1,i2,i3,i4);
}

```

The function prints the following:

```
i1=0 i2=1 i3=0 i4=0
```

int vc_isMemory(vc_handle)

This routine returns a 1 value if the vc_handle is to a memory. It returns a 0 value for a bit or reg that is not a memory. For example, using the Verilog code from the previous example and the following C/C++ function:

```
#include <stdio.h>
#include "DirectC.h"

scalarfinder(vc_handle h1, vc_handle h2, vc_handle h3,
vc_handle h4)
{
    int i1 = vc_isMemory(h1),
        i2 = vc_isMemory(h2),
        i3 = vc_isMemory(h3),
        i4 = vc_isMemory(h4);
    printf("\ni1=%d i2=%d i3=%d i4=%d\n\n",i1,i2,i3,i4);
}
```

The function prints the following:

```
i1=0 i2=0 i3=1 i4=1
```

int vc_is4state(vc_handle)

This routine returns a 1 value if the vc_handle is to a reg or memory that simulates with four states. It returns a 0 value for a bit or a memory that simulates with two states. For example, the following Verilog code uses metacomments to specify four- and two-state simulation:

```
extern void statefinder (input reg r1,
                        input reg [1:0] r2,
                        input reg [1:0] array [1:0] r3,
                        input reg array [1:0] r4,
                        input bit r5,
                        input bit [1:0] r6,
                        input bit [1:0] array [1:0] r7,
                        input bit array [1:0] r8);

module top;
reg /*4value*/ r1;
reg /*4value*/ [1:0] r2;
reg /*4value*/ [1:0] r3 [1:0];
reg /*4value*/ r4 [1:0];
reg /*2value*/ r5;
reg /*2value*/ [1:0] r6;
reg /*2value*/ [1:0] r7 [1:0];
reg /*2value*/ r8 [1:0];
initial
statefinder(r1,r2,r3,r4,r5,r6,r7,r8);
endmodule
```

The C/C++ function that calls the `vc_is4state` routine is as follows:

```
#include <stdio.h>
#include "DirectC.h"

statefinder(vc_handle h1, vc_handle h2, vc_handle h3,
            vc_handle h4, vc_handle h5, vc_handle h6,
            vc_handle h7, vc_handle h8)
{
    printf("\nThe vc_handles to 4state are:");
    printf("\nh1=%d h2=%d h3=%d h4=%d\n\n",
           vc_is4state(h1), vc_is4state(h2),
           vc_is4state(h3), vc_is4state(h4));
    printf("\nThe vc_handles to 2state are:");
    printf("\nh5=%d h6=%d h7=%d h8=%d\n\n",
           vc_is4state(h5), vc_is4state(h6),
           vc_is4state(h7), vc_is4state(h8));
}
```

The function prints the following:

```
The vc_handles to 4state are:
h1=1 h2=1 h3=1 h4=1
```

```
The vc_handles to 2state are:
h5=0 h6=0 h7=0 h8=0
```

int vc_is2state(vc_handle)

This routine does the opposite of the `vc_is4state` routine. For example, using the Verilog code from the previous example and the following C/C++ function:

```
#include <stdio.h>
#include "DirectC.h"

statefinder(vc_handle h1, vc_handle h2, vc_handle h3,
            vc_handle h4, vc_handle h5, vc_handle h6,
            vc_handle h7, vc_handle h8)
{
    printf("\nThe vc_handles to 4state are:");
    printf("nh1=%d h2=%d h3=%d h4=%d\n\n",
          vc_is2state(h1), vc_is2state(h2),
          vc_is2state(h3), vc_is2state(h4));
    printf("\nThe vc_handles to 2state are:");
    printf("nh5=%d h6=%d h7=%d h8=%d\n\n",
          vc_is2state(h5), vc_is2state(h6),
          vc_is2state(h7), vc_is2state(h8));
}
```

The function prints the following:

```
The vc_handles to 4state are:
h1=0 h2=0 h3=0 h4=0
```

```
The vc_handles to 2state are:
h5=1 h6=1 h7=1 h8=1
```

int vc_is4stVector(vc_handle)

This routine returns a 1 value if the vc_handle is to a vector reg. It returns a 0 value if the vc_handle is to a scalar reg, scalar or vector bit, or memory. For example, using the Verilog code from the previous example, and the following C/C++ function:

```
#include <stdio.h>
#include "DirectC.h"

statefinder(vc_handle h1, vc_handle h2,
            vc_handle h3, vc_handle h4,
            vc_handle h5, vc_handle h6,
            vc_handle h7, vc_handle h8)
{
    printf("\nThe vc_handle to a 4state Vector is:");
    printf("\nh2=%d \n\n", vc_is4stVector(h2));
    printf("\nThe vc_handles to 4state scalars or
          memories and 2state are:");
    printf("\nh1=%d h3=%d h4=%d h5=%d h6=%d h7=%d h8=%d\n\n",
           vc_is4stVector(h1), vc_is4stVector(h3),
           vc_is4stVector(h4), vc_is4stVector(h5),
           vc_is4stVector(h6), vc_is4stVector(h7),
           vc_is4stVector(h8));
}
```

The function prints the following:

```
The vc_handle to a 4state Vector is:
h2=1
```

```
The vc_handles to 4state scalars or
memories and 2state are:
h1=0 h3=0 h4=0 h5=0 h6=0 h7=0 h8=0
```

int vc_is2stVector(vc_handle)

This routine returns a 1 value if the vc_handle is to a vector bit. It returns a 0 value if the vc_handle is to a scalar bit, scalar or vector reg, or to a memory. For example, using the Verilog code from the previous example and the following C/C++ function:

```
#include <stdio.h>
#include "DirectC.h"

statefinder(vc_handle h1, vc_handle h2,
            vc_handle h3, vc_handle h4,
            vc_handle h5, vc_handle h6,
            vc_handle h7, vc_handle h8)
{
    printf("\nThe vc_handle to a 2state Vector is:");
    printf("\nh6=%d \n\n", vc_is2stVector(h6));
    printf("\nThe vc_handles to 2state scalars or
          memories and 4state are:");
    printf("\nh1=%d h2=%d h3=%d h4=%d h5=%d h7=%d h8=%d\n\n",
           vc_is2stVector(h1), vc_is2stVector(h2),
           vc_is2stVector(h3), vc_is2stVector(h4),
           vc_is2stVector(h5), vc_is2stVector(h7),
           vc_is2stVector(h8));
}
```

The function prints the following:

```
The vc_handle to a 2state Vector is:
h6=1
```

```
The vc_handles to 2state scalars or
memories and 4state are:
h1=0 h2=0 h3=0 h4=0 h5=0 h7=0 h8=0
```

int vc_width(vc_handle)

Returns the width of a vc_handle. For example:

```
void memcheck_int(vc_handle h)
{
    int i;

    int mem_size = vc_arraySize(h);

    /* determine minimal needed width, assuming signed int */
    for (i=0; (1 << i) < (mem_size-1); i++) ;

    if (vc_width(h) < (i+1)) {
        printf("Register too narrow to be assigned %d\n",
               (mem_size-1));
        return;
    }

    for(i=0;i<8;i++) {
        vc_putMemoryInteger(h,i,i*4);
        printf("memput : %d\n",i*4);
    }
    for(i=0;i<8;i++) {
        printf("memget:: %d \n",vc_getMemoryInteger(h,i));
    }
}
```

int vc_arraySize(vc_handle)

Returns the number of elements in a memory or multi-dimensional array. The previous example also shows a usage of vc_arraySize().

scalar vc_getScalar(vc_handle)

Returns the value of a scalar reg or bit. For example:

```
void rotate_scalars(vc_handle h1, vc_handle h2, vc_handle
```

```

h3)
{
    scalar a;

    a = vc_getScalar(h1);
    vc_putScalar(h1, vc_getScalar(h2));
    vc_putScalar(h2, vc_getScalar(h3));
    vc_putScalar(h3, a);
    return;
}

```

void vc_putScalar(vc_handle, scalar)

Passes the value of a scalar reg or bit to a `vc_handle` by reference.
The previous example also shows a usage of `vc_putScalar()`.

char vc_toChar(vc_handle)

Returns the 0, 1, x, or z character. For example:

```

void print_scalar(vc_handle h) {
    printf("%c", vc_toChar(h));
    return;
}

```

int vc_toInteger(vc_handle)

Returns an int value for a `vc_handle` to a scalar bit or a vector bit of 32 bits or less. For a vector reg or a vector bit with more than 32 bits this routine returns a 0 value and displays the following warning message:

```
DirectC interface warning: 0 returned for 4-state value
(vc_toInteger)
```

The following is an example of Verilog code that calls a C/C++ function that uses this routine:

```
extern void rout1 (input bit onebit, input bit [7:0] mobits);

module top;
reg /*2value*/ onebit;
reg /*2value*/ [7:0] mobits;
initial
begin
rout1(onebit,mobits);
onebit=1;
mobits=128;
rout1(onebit,mobits);
end
endmodule
```

Notice that the function declaration specifies that the parameters are of type bit. It includes metacomments for two-state simulation in the declaration of reg onebit and mobits. There are two calls to the function rout1, before and after values are assigned in this Verilog code.

The following C/C++ function uses this routine:

```
#include <stdio.h>
#include "DirectC.h"

void rout1 (vc_handle onebit, vc_handle mobits)
{
printf("\n\nonebit is %d mobits is %d\n\n",
      vc_toInteger(onebit), vc_toInteger(mobits));
}
```

This function prints the following:

```
onebit is 0 mobits is 0
```

```
onebit is 1 mobits is 128
```

char *vc_toString(vc_handle)

Returns a string that contains the 1, 0, x, and z characters. For example:

```
extern void vector_printer (input reg [7:0] r1);

module test;
reg [7:0] r1,r2;

initial
begin
#5 r1 = 8'bzx01zx01;
#5 vector_printer(r1);
#5 $finish;
end
endmodule

void vector_printer (vc_handle h)
{
vec32 b,*c;
c=vc_4stVectorRef(h) ;
b=*c;
printf("\n b is %x[control] %x[data]\n\n",b.c,b.d) ;
printf("\n b is %s \n\n",vc_toString(h)) ;
}
```

In this example, a vector reg is assigned a value that contains x and z values, as well as, 1 and 0 values. In the abstract access C/C++ function, there are two ways of displaying the value of the reg:

- Recognize that type `vec32` is defined as follows in the `DirectC.h` file:

```
typedef struct {U c; U d;} vec32;
```

In machine memory, there are control, as well as, data bits for Verilog data to differentiate X from 1 and Z from 0 data, so there are `c` (control) and `d` (data) data variables in the structure and you must specify which variable when you access the `vec32` type.

- Use the `vc_toString` routine to display the value of the reg that contains X and Z values.

This example displays:

```
b is cc[control 55 [data]
```

```
b is zx01zx01
```

char *vc_toStringF(vc_handle, char)

Returns a string that contains the 1, 0, x, and z characters and allows you to specify the format or radix for the display. The `char` parameter can be '`b`', '`o`', '`d`', or '`x`'.

So, if we modify the C/C++ function in the previous example, it is as follows:

```
void vector_printer (vc_handle h)
{
vec32 b,*c;
c=vc_4stVectorRef(h) ;
b=*c;
printf("\n b is %s \n\n",vc_toStringF(h,'b'));
printf("\n b is %s \n\n",vc_toStringF(h,'o'));
printf("\n b is %s \n\n",vc_toStringF(h,'d'));
printf("\n b is %s \n\n",vc_toStringF(h,'x'));
}
```

This example now displays:

```
b is zx01zx01
```

```
b is XZX
```

```
b is X
```

```
b is XX
```

void vc_putReal(vc_handle, double)

Passes by reference a real (double) value to a vc_handle. For example:

```
void get_PI(vc_handle h)
{
    vc_putReal(h, 3.14159265);
}
```

double vc_getReal(vc_handle)

Returns a real (double) value from a vc_handle. For example:

```
void print_real(vc_handle h)
{
    printf("[print_real] %f\n", vc_getReal(h));
}
```

void vc_putValue(vc_handle, char *)

This function passes, by reference, through the vc_handle, a value represented as a string containing the 0, 1, x, and z characters. For example:

```
extern void check_vc_putvalue(output reg [] r1);

module tester;
reg [31:0] r1;

initial
begin
check_vc_putvalue(r1);
$display("r1=%0b",r1);
$finish;
end
endmodule
```

In this example, the C/C++ function is declared in the Verilog code specifying that the function passes a value to a four-state reg (and, therefore, can hold X and Z values).

```
#include <stdio.h>
#include "DirectC.h"

void check_vc_putvalue(vc_handle h)
{
    vc_putValue(h, "10xz");
}
```

The `vc_putValue` routine passes the string "10xz" to the reg `r1` through the `vc_handle`. The Verilog code displays:

```
r1=10xz
```

void vc_putValueF(vc_handle, char *, char)

This function passes by reference, through the `vc_handle`, a value for which you specify a radix with the third parameter. The valid radices are 'b', 'o', 'd', and 'x'. For example the following Verilog code declares a function named `assigner` that uses this routine:

```
extern void assigner (output reg [31:0] r1,
                      output reg [31:0] r2,
                      output reg [31:0] r3,
                      output reg [31:0] r4);

module test;
reg [31:0] r1,r2,r3,r4;
initial
begin
assigner(r1,r2,r3,r4);
$display("r1=%0b in binary r1=%0d in decimal\n",r1,r1);
$display("r2=%0o in octal r2 =%0d in decimal\n",r2,r2);
$display("r3=%0d in decimal r3=%0b in binary\n",r3,r3);
$display("r4=%0h in hex r4= %0d in decimal\n\n",r4,r4);
$finish;
end
endmodule
```

The following is the C/C++ function:

```
#include <stdio.h>
#include "DirectC.h"

void assigner (vc_handle h1, vc_handle h2, vc_handle h3,
```

```
vc_handle h4)
{
    vc_putValueF(h1, "10",'b');
    vc_putValueF(h2, "11",'o');
    vc_putValueF(h3, "10",'d');
    vc_putValueF(h4, "aff",'x');
}
```

The Verilog code displays the following:

```
r1=10 in binary r1=2 in decimal  
r2=11 in octal r2 =9 in decimal  
r3=10 in decimal r3=1010 in binary  
r4=aff in hex r4= 2815 in decimal
```

void vc_putPointer(vc_handle, void*)
void *vc_getPointer(vc_handle)

These functions pass a generic type of pointer or string to a `vc_handle` by reference. Do not use these functions for passing Verilog data (the values of Verilog signals). Use them for passing C/C++ data instead. `vc_putPointer` passes this data by reference to Verilog and `vc_getPointer` receives this data in a pass by reference from Verilog. You can also use these functions for passing Verilog strings.

For example:

```
extern void passback(output string, input string);
extern void printer(input pointer);

module top;
reg [31:0] r2;
initial
begin
passback(r2, "abc");
printer(r2);
end
endmodule
```

This Verilog code passes the string "abc" to the `passback` C/C++ function by reference, and that function passes it by reference to reg `r2`. The Verilog code then passes it by reference to the C/C++ function `printer` from reg `r2`.

```
passback(vc_handle h1, vc_handle h2)
{
    vc_putPointer(h1, vc_getPointer(h2));
}

printer(vc_handle h)
{
    printf("Procedure printer prints the string value %s\n\n",
           vc_getPointer (h));
}
```

The function named `printer` prints the following:

```
Procedure printer prints the string value abc
```

void vc_StringToVector(char *, vc_handle)

Converts a C string (a pointer to a sequence of ASCII characters terminated with a null character) into a Verilog string (a vector with 8-bit groups representing characters). For example:

```
extern "C" stringFullPath(string filename);
// find full path to the file
// C string obtained from C domain

extern "A" void s2v(string, output reg[] );
// string-to-vector
// wrapper for vc_StringToVector().

`define FILE_NAME_SIZE 512

module Test;
    reg [`FILE_NAME_SIZE*8:1] file_name;
    // this file_name will be passed to the Verilog code that
    expects
    // a Verilog-like string
    .
    .
    .
    initial begin
        s2vFullPath("myStimulusFile"), file_name); // C-string to
        Verilog-string
        // bits of 'file_name' represent now 'Verilog string'
    end
    .
    .
    .
endmodule
```

The C code is as follows:

```
void s2v(vc_handle hs, vc_handle hv) {
    vc_StringToVector((char *)vc_getPointer(hs), hv);
}
```

void vc_VectorToString(vc_handle, char *)

Converts a vector value to a string value.

int vc_getInteger(vc_handle)

Same as vc_toInteger.

void vc_putInteger(vc_handle, int)

Passes an int value by reference through a vc_handle to a scalar reg or bit or a vector bit that is 32 bits or less. For example:

```
void putter (vc_handle h1, vc_handle h2, vc_handle h3,
vc_handle h4)
{
    int a,b,c,d;
    a=1;
    b=2;
    c=3;
    d=9999999;
```



```
    vc_putInteger(h1,a);
    vc_putInteger(h2,b);
    vc_putInteger(h3,c);
    vc_putInteger(h4,d);
}
```

vec32 *vc_4stVectorRef(vc_handle)

Returns a vec32 pointer to a four-state vector. Returns NULL if the specified vc_handle is not to a four-state vector reg. For example:

```
typedef struct vector_descriptor {
    int width; /* number ofbits */
    int is4stte; /* TRUE/FALSE */
} VD;
```



```
void WriteVector(vc_handle file_handle, vc_handle a_vector)
```

```

{
    FILE *fp;
    int n, size;
    vec32 *v;
    VD      vd;
    fp = vc_getPointer(file_handle);

    /* write vector's size and type */
    vd.is4state = vc_is4stVector(a_vector);
    vd.width = vc_width(a_vector);
    size = (vd.width + 31) >> 5; /* number of 32-bit chunks */
    /* printf("writing: %d bits, is 4 state: %d, #chunks:
       %d\n", vd.width, vd.is4state, size); */
    n = fwrite(&vd, sizeof(vd), 1, fp);
    if (n != 1) {
        printf("Error: write failed.\n");
    }

    /* write the vector into a file; vc_*stVectorRef
       is a pointer to the actual Verilog vector */
    if (vc_is4stVector(a_vector)) {
        n = fwrite(vc_4stVectorRef(a_vector), sizeof(vec32),
                   size, fp);
    } else {
        n = fwrite(vc_2stVectorRef(a_vector), sizeof(U),
                   size, fp);
    }
    if (n != size) {
        printf("Error: write failed for vector.\n");
    }
}

```

U *vc_2stVectorRef(vc_handle)

Returns a U pointer to a bit vector that is larger than 32 bits. If you specify a short bit vector (32 bits or fewer) this routine returns a NULL value. For example:

In this example, the Verilog code declares a 32-bit vector bit, `r1`, and a 33-bit vector bit, `r2`. The values of both are passed to the C/C++ function `big_2state`.

When we pass the short bit vector `r1` to `vc_2stVectorRef`, it returns a null value because it has fewer than 33 bits. This is not the case when we pass bit vector `r2` because it has more than 32 bits.

Notice that from right to left, the first 32 bits of `r2` have a value of 2 and the MSB 33rd bit has a value of 1. This is significant in how the C/C++ stores this data.

```
#include <stdio.h>
#include "DirectC.h"

big_2state(vc_handle h1, vc_handle h2)
{
    U u1,*up1,u2,*up2;
    int i;
    int size;

    up1=vc_2stVectorRef(h1);
    up2=vc_2stVectorRef(h2);
    if (up1){ /* check for the null value returned to up1 */
        u1=*up1;} else{
        u1=0;
        printf("\nShort 2 state vector passed to up1\n");
    }
    if (up2){ /* check for the null value returned to up2 */
        size = vc_width (h2); /* to find out the number of bits */
        /* in h2 */
        printf("\n width of h2 is %d\n",size);
        size = (size + 31) >> 5; /* to get number of 32-bit chunks */
        printf("\n the number of chunks needed for h2 is %d\n\n",
               size);
        printf("loading into u2");
        for(i = size - 1; i >= 0; i--){
            u2=up2[i]; /* load a chunk of the vector */
            printf(" %x",up2[i]);
        }
        printf("\n");
    } else{
        u2=0;
        printf("\nShort 2 state vector passed to up2\n");
    }
}
```

In this example, the short bit vector is passed to the `vc_2stVectorRef` routine, so it returns a null value to pointer `up1`. Then the long bit vector is passed to the `vc_2stVectorRef` routine, so it returns a pointer to the Verilog data for vector bit `r2` to pointer `up2`.

It checks for the null value in `up1`. If it doesn't have a null value, whatever it points to is passed to `u1`. If it does have a null value, the function prints a message about the short bit vector. In this example, you can expect it to print this message.

Still later in the function, it checks for the null value in `up2` and the size of the long bit vector that is passed to the second parameter. Then, because Verilog values are stored in 32-bit chunks in C/C++, the function finds out how many chunks are needed to store the long bit vector. It then loads one chunk at a time into `u2` and prints the chunk starting with the most significant bits. This function displays the following:

```
Short 2 state vector passed to up1
width of h2 is 33
the number of chunks needed for h2 is 2
loading into u2 1 2
```

```
void vc_get4stVector(vc_handle, vec32 *)
void vc_put4stVector(vc_handle, vec32 *)
```

Passes a four-state vector by reference to a `vc_handle` to and from an array in C/C++ function. `vc_get4stVector` receives the vector from Verilog and passes it to the array and `vc_put4stVector` passes the array to Verilog.

These routines work only if there are enough elements in the array for all the bits in the vector. The array must have an element for every 32 bit in the vector plus an additional element for any remaining bits. For example:

```
extern void copier (input reg [67:0] r1,
                    output reg [67:0] r2);

module top;

reg [67:0] r1,r2;

initial
begin
    r1 [67:65] = 3'b111;
    r1 [64:33] = 32'bzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz;
    r1 [32:0] = 32'b00000000000000000000000000000000;
    copier(r1,r2);
    $display("r1=%0b\n",r1);
    $display("r2=%0b\n",r2);

end
endmodule
```

In this example, there are two 68-bit regs. Values are assigned to all the bits of one reg and both of these regs are parameters to the C/C++ function named `copier`.

```
copier(vc_handle h1, vc_handle h2)
{
vec32 holder[3];
vc_get4stVector(h1,holder);
vc_put4stVector(h2,holder);
}
```

This function declares a `vec32` array of three elements named `holder`. It uses three elements because its parameters are 68-bit regs so we need an element for every 32 bits and one more for the remaining four bits.

The Verilog code displays the following:

```
void vc_get2stVector(vc_handle, U *)
void vc_put2stVector(vc_handle, U *)
```

Passes a two-state vector by reference to a vc_handle to and from an array in C/C++ function. vc_get2stVector receives the vector from Verilog and passes it to the array and vc_put4stVector passes the array to Verilog.

These routines, just like the `vc_get4stVector` and `vc_put4stVector` routines, work only if there are enough elements in the array for all the bits in the vector. The array must have an element for every 32 bit in the vector plus an additional element for any remaining bits.

The only differences between these routines and the `vc_get4stVector` and `vc_put4stVector` routines are the type of data they pass, two- or four-state simulation values, and the type you declare for the array in the C/C++ function.

UB *vc_MemoryRef(vc_handle)

Returns a pointer of type UB that points to a memory in Verilog. For example:

```
extern void mem_doer ( input reg [1:0] array [3:0]
                      memory1, output reg [1:0] array
                      [31:0] memory2) ;

module top;
reg [1:0] memory1 [3:0];
reg [1:0] memory2 [31:0];
initial
begin
memory1 [3] = 2'b11;
memory1 [2] = 2'b10;
memory1 [1] = 2'b01;
memory1 [0] = 2'b00;
mem_doer(memory1,memory2);
$display("memory2[31]=%0d",memory2[31]);
end
endmodule
```

In this example, we declare two memories, one with 4 addresses, `memory1`, the other with 32 addresses, `memory2`. We assign values to the addresses of `memory1`, and then pass both memories to the C/C++ function `mem_doer`.

```
#include <stdio.h>
#include "DirectC.h"

void mem_doer(vc_handle h1, vc_handle h2)
{
    UB *p1, *p2;
    int i;

    p1 = vc_MemoryRef(h1);
    p2 = vc_MemoryRef(h2);

    for (i = 0; i < 8; i++) {
        memcpy(p2, p1, 8);
        p2 += 8;
    }
}
```

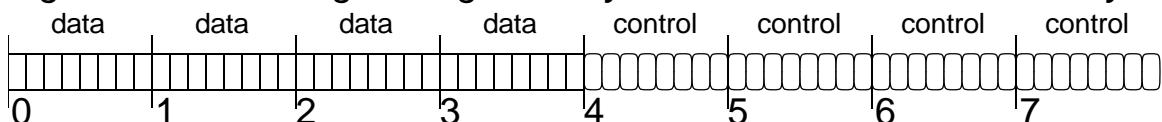
The purpose of the C/C++ function `mem_doer` is to copy the four elements in Verilog memory `memory1` into the 32 elements of `memory2`.

The `vc_MemoryRef` routines return pointers to the Verilog memories and the machine memory locations they point to are also pointed to by pointers `p1` and `p2`. Pointer `p1` points to the location of Verilog memory `memory1`, and `p2` points to the location of Verilog memory `memory2`.

The function uses a for loop to copy the data from Verilog memory `memory1` to Verilog memory `memory2`. It uses the standard `memcpy` function to copy a total of 64 bytes by copying eight bytes eight times.

This example copies a total of 64 bytes because each element of `memory2` is only two bits wide, but for every eight bits in an element in machine memory there are two bytes, one for data and another for control. The bits in the control byte specify whether the data bit with a value of 0 is actually 0 or Z, or whether the data bit with a value of 1 is actually 1 or X.

Figure 23-4 Storing Verilog Memory Elements in Machine Memory



In an element in a Verilog memory, for each eight bits in the element there is a data byte and a control byte with an additional set of bytes for a remainder bit. So, if a memory had 9 bits it would need two data bytes and two control bytes. If it had 17 bits it would need three data bytes and three control bytes. All the data bytes precede the control bytes.

Therefore, `memory1` needs 8 bytes of machine memory (four for data and four for control) and `memory2` needs 64 bytes of machine memory (32 for data and 32 for control). Therefore, the C/C++ function needs to copy 64 bytes.

The Verilog code displays the following:

```
memory2 [31] =3
```

UB *vc_MemoryElemRef(vc_handle, U indx)

Returns a pointer to an element (word, address or index) of a Verilog memory. You specify the `vc_handle` of the memory and the element. For example:

```
extern void mem_elem_doer( inout reg [25:1] array [3:0]
```

```
memory1) ;

module top;
reg [25:1] memory1 [3:0];
initial
begin
memory1 [0] = 25'b00000000xxxxxxxx11111111;
$display("memory1 [0] = %0b\n", memory1[0]);
mem_add_doer(memory1);
$display("\nmemory1 [3] = %0b", memory1[3]);
end
endmodule
```

In this example, there is a Verilog memory with four addresses, each element has 25 bits. This means that the Verilog memory needs eight bytes of machine memory because there is a data byte and a control byte for every eight bits in an element, with an additional data and control byte for any remainder bits.

In this example, in element 0 the 25 bits are assigned, from right to left, eight 1 bits, eight unknown x bits, eight 0 bits, and one high impedance z bit.

```
#include <stdio.h>
#include "DirectC.h"

void mem_elem_doir(vc_handle h)
{
    U indx;
    UB *p1, *p2, t [8];

    indx = 0;
    p1 = vc_MemoryElemRef(h, indx);
    indx = 3;
    p2 = vc_MemoryElemRef(h, indx);
    memcpy(p2,p1,8);

    memcpy(t,p2,8);
    printf(" %d from t[0], %d from t[1]\n",
           (int)t[0], (int) t[1]);
    printf(" %d from t[2], %d from t[3]\n",
           (int)t[2], (int) t[3]);
    printf(" %d from t[4], %d from t[5]\n",
           (int)t[4], (int)t[5]);
    printf(" %d from t[6], %d from t[7]\n",
           (int)t[6], (int)t[7]);
}

}
```

C/C++ function `mem_elem_doir` uses the `vc_MemoryElemRef` routine to return pointers to addresses 0 and 3 in Verilog memory1 and pass them to UB pointers `p1` and `p2`. The standard `memcpy` routine then copies the eight bytes for address 0 to address 3.

The remainder of the function is additional code to show you data and control bytes. The eight bytes pointed to by p2 are copied to array t and then the elements of the array are printed.

The combined Verilog and C/C++ code displays the following:

```
memory1 [0] = z0000000xxxxxxxxx11111111  
  
255 from t[0], 255 from t[1]  
0 from t[2], 0 from t[3]  
0 from t[4], 255 from t[5]  
0 from t[6], 1 from t[7]  
  
memory1 [3] = z0000000xxxxxxxxx11111111
```

As you can see, function `mem_elem_doer` passes the contents of the Verilog memory `memory1` element 0 to element 3.

In array t, the elements contain the following:

- [0] The data bits for the eight 1 values assigned to the element.
- [1] The data bits for the eight X values assigned to the element
- [2] The data bits for the eight 0 values assigned to the element
- [3] The data bit for the Z value assigned to the element
- [4] The control bits for the eight 1 values assigned to the element
- [5] The control bits for the eight X values assigned to the element
- [6] The control bits for the eight 0 values assigned to the element
- [7] The control bit for the Z value assigned to the element

scalar vc_getMemoryScalar(vc_handle, U indx)

Returns the value of a one-bit memory element. For example:

```
extern void bitflipper (inout reg array [127:0] mem1);  
  
module test;  
reg mem1 [127:0];  
initial  
begin  
mem1 [0] = 1;  
$display("mem1[0]=%0d",mem1[0]);  
bitflipper(mem1);  
$display("mem1[0]=%0d",mem1[0]);  
$finish;  
end  
endmodule
```

In this example of Verilog code, we declare a memory with 128 one-bit elements, assign a value to element 0, and display its value before and after we call a C/C++ function named bitflipper.

```
#include <stdio.h>  
#include "DirectC.h"  
  
void bitflipper(vc_handle h)  
{  
scalar holder=vc_getMemoryScalar(h, 0);  
holder = ! holder;  
vc_putMemoryScalar(h, 0, holder);  
}
```

In this example, we declare a variable of type scalar, named `holder`, to hold the value of the one-bit Verilog memory element. The routine `vc_getMemoryScalar` returns the value of the element to the variable. The value of `holder` is inverted and then

the variable is included as a parameter in the `vc_putMemoryScalar` routine to pass the value to that element in the Verilog memory.

The Verilog code displays the following:

```
mem[0]=1  
mem[0]=0
```

void vc_putMemoryScalar(vc_handle, U indx, scalar)

Passes a value of type scalar to a Verilog memory element. You specify the memory by `vc_handle` and the element by the `indx` parameter. This routine is used in the previous example.

int vc_getMemoryInteger(vc_handle, U indx)

Returns the integer equivalent of the data bits in a memory element whose bit-width is 32 bits or less. For example:

```
extern void mem_elem_halver (inout reg [] array [] memX);  
  
module test;  
reg [31:0] mem1 [127:0];  
reg [7:0] mem2 [1:0];  
initial  
begin  
mem1 [0] = 999;  
mem2 [0] = 8'b1111xxxx;  
$display("mem1[0]=%0d",mem1[0]);  
$display("mem2[0]=%0d",mem2[0]);  
mem_elem_halver(mem1);  
mem_elem_halver(mem2);  
$display("mem1[0]=%0d",mem1[0]);  
$display("mem2[0]=%0d",mem2[0]);  
$finish;  
end  
endmodule
```

In this example, when the C/C++ function is declared on our Verilog code it does not specify a bit-width or element range for the inout argument to the `mem_elem_halver` C/C++ function, because in the Verilog code we call the C/C++ function twice, with a different memory each time and these memories have different bit widths and different element ranges.

Notice that we assign a value that included X values to the 0 element in memory `mem2`.

```
#include <stdio.h>
#include "DirectC.h"

void mem_elem_halver(vc_handle h)
{
    int i = vc_getMemoryInteger(h, 0);
    i = i/2;
    vc_putMemoryInteger(h, 0, i);
}
```

This C/C++ function inputs the value of an element and then outputs half that value. The `vc_getMemoryInteger` routine returns the integer equivalent of the element you specify by `vc_handle` and index number, to an `int` variable `i`. The function halves the value in `i`. Then the `vc_putMemoryInteger` routine passes the new value by value to the specified memory element.

The Verilog code displays the following before the C/C++ function is called twice with the different memories as the arguments:

```
mem1 [0] =999
mem2 [0] =X
```

Element `mem2[0]` has an X value because half of its binary value is x and the value is displayed with the %d format specification and, in this example, a partially unknown value is just an unknown value. After the second call of the function, the Verilog code displays:

```
mem1[1]=499  
mem2[0]=127
```

This occurs because before calling the function, `mem1[0]` had a value of 999, and after the call it has a value of 499 which is as close as it can get to half the value with integer values.

Before calling the function, `mem2[0]` had a value of 8'b1111xxxx, but the data bits for the element would all be 1s (11111111). It's the control bits that specify 1 from x and this routine only deals with the data bits. So, the `vc_getMemoryInteger` routine returned an integer value of 255 (the integer equivalent of the binary 11111111) to the C/C++ function, which is why the function outputs the integer value 127 to `mem2[0]`.

void vc_putMemoryInteger(vc_handle, U indx, int)

Passes an integer value to a memory element that is 32 bits or fewer. You specify the memory by `vc_handle` and the element by the `indx` argument. This routine is used in the previous example.

void vc_get4stMemoryVector(vc_handle, U indx, vec32 *)

Copies the value in an Verilog memory element to an element in an array. This routine copies both the data and control bytes. It copies them into an array of type `vec32` which is defined as follows:

```
typedef struct { U c; U d; } vec32;
```

Therefore, type `vec32` has two members, `c` and `d`, for control and data information. This routine always copies to the 0 element of the array. For example:

```
extern void mem_elem_copier (inout reg [] array [] memX);  
  
module test;  
reg [127:0] mem1 [127:0];  
reg [7:0] mem2 [64:0];  
initial  
begin  
mem1 [0] = 999;  
mem2 [0] = 8'b0000000z;  
$display("mem1[0]=%0d",mem1[0]);  
$display("mem2[0]=%0d",mem2[0]);  
mem_elem_copier(mem1);  
mem_elem_copier(mem2);  
$display("mem1[32]=%0d",mem1[32]);  
$display("mem2[32]=%0d",mem2[32]);  
$finish;  
end  
endmodule
```

In the Verilog code, a C/C++ function is declared that is called twice. Notice the value assigned to `mem2 [0]`. The C/C++ function copies the values to another element in the memory.

```
#include <stdio.h>  
#include "DirectC.h"  
  
void mem_elem_copier(vc_handle h)  
{  
vec32 holder[1];  
vc_get4stMemoryVector(h,0,holder);  
vc_put4stMemoryVector(h,32,holder);  
printf(" holder[0].d is %d holder[0].c is %d\n\n",  
      holder[0].d,holder[0].c);  
}
```

This C/C++ function declares an array of type `vec32`. We must declare an array for this type, but as shown here, we specify that it have only one element. The `vc_get4stMemoryVector` routine copies the data from the Verilog memory element (in this example, specified as the 0 element) to the 0 element of the `vec32` array. It always copies to the 0 element. The `vc_put4stMemoryVector` routine copies the data from the `vec32` array to the Verilog memory element (in this case, element 32).

The call to `printf` is to show you how the Verilog data is stored in element 0 of the `vec32` array.

The Verilog and C/C++ code display the following:

```
mem1[0]=999
mem2[0]=Z
holder[0].d is 999 holder[0].c is 0

holder[0].d is 768 holder[0].c is 1

mem1[32]=999
mem2[32]=Z
```

As you can see, the function does copy the Verilog data from one element to another in both memories. When the function is copying the 999 value, the `c` (control) member has a value of 0; when it is copying the 8'b0000000z value, the `c` (control) member has a value of 1 because one of the control bits is 1, the rest are 0.

void vc_put4stMemoryVector(vc_handle, U indx, vec32 *)

Copies Verilog data from a `vec32` array to a Verilog memory element. This routine is used in the previous example.

void vc_get2stMemoryVector(vc_handle, U indx, U *)

Copies the data bytes, but not the control bytes, from a Verilog memory element to an array in your C/C++ function. For example, if you use the Verilog code from the previous example, but simulate in two-state and use the following C/C++ code:

```
#include <stdio.h>
#include "DirectC.h"

void mem_elem_copier(vc_handle h)
{
    U holder[1];
    vc_get2stMemoryVector(h, 0, holder);
    vc_put2stMemoryVector(h, 32, holder);

}
```

The only difference here is that we declare the array to be of type U instead and we do not copy the control bytes, because there are none in two-state simulation.

void vc_put2stMemoryVector(vc_handle, U indx, U *)

Copies Verilog data from a U array to a Verilog memory element. This routine is used in the previous example.

void vc_putMemoryValue(vc_handle, U idx, char *)

This routine works like the vc_putValue routine except that is for passing values to a memory element instead of to a reg or bit. You enter an argument to specify the element (index) to which you want the routine to pass the value. For example:

```
#include <stdio.h>
#include "DirectC.h"

void check_vc_putvalue(vc_handle h)
{
    vc_putMemoryValue(h, 0, "10xz");
}
```

void vc_putMemoryValueF(vc_handle, U idx, char, char *)

This routine works like the vc_putValueF routine except that it is for passing values to a memory element instead of to a reg or bit. You enter an argument to specify the element (index) to which you want the routine to pass the value. For example:

```
#include <stdio.h>
#include "DirectC.h"

void assigner (vc_handle h1, vc_handle h2, vc_handle h3,
vc_handle h4)
{
    vc_putMemoryValueF(h1, 0, "10", 'b');
    vc_putMemoryValueF(h2, 0, "11", 'o');
    vc_putMemoryValueF(h3, 0, "10", 'd');
    vc_putMemoryValueF(h4, 0, "aff", 'x');
```

char *vc_MemoryString(vc_handle, U indx)

This routine works like the vc_toString routine except that it used is for passing values to/from memory elements instead of to a reg or bit. You enter an argument to specify the element (index) whose value you want the routine to pass. For example:

```
extern void    memcheck_vec(inout reg[] array[]) ;

module top;
reg [0:7] mem[0:7];
integer i;

initial
begin
    for(i=0;i<8;i=i+1) begin
        mem[i] = 8'b00000111;
        $display("Verilog code says \\"mem [%0d] = %0b\\\"", i,mem[i]);
    end

    memcheck_vec(mem);
end

endmodule
```

The C/C++ function that calls vc_MemoryString is as follows:

```
#include <stdio.h>
#include "DirectC.h"

void memcheck_vec(vc_handle h)
{
    int i;

    for(i= 0; i<8;i++) {
        printf("C/C++ code says \"mem [%d] is %s
```

```
\\"\\n", i, vc_MemoryString(h, i));  
    }  
}
```

The Verilog and C/C++ code display the following:

```
Verilog code says "mem [0] = 111"  
Verilog code says "mem [1] = 111"  
Verilog code says "mem [2] = 111"  
Verilog code says "mem [3] = 111"  
Verilog code says "mem [4] = 111"  
Verilog code says "mem [5] = 111"  
Verilog code says "mem [6] = 111"  
Verilog code says "mem [7] = 111"  
C/C++ code says "mem [0] is 00000111 "  
C/C++ code says "mem [1] is 00000111 "  
C/C++ code says "mem [2] is 00000111 "  
C/C++ code says "mem [3] is 00000111 "  
C/C++ code says "mem [4] is 00000111 "  
C/C++ code says "mem [5] is 00000111 "  
C/C++ code says "mem [6] is 00000111 "  
C/C++ code says "mem [7] is 00000111 "
```

char *vc_MemoryStringF(vc_handle, U indx, char)

This routine works like the vc_MemoryString function except that you specify a radix with the third parameter. The valid radices are 'b', 'o', 'd', and 'x'. For example:

```
extern void    memcheck_vec(inout reg[] array[]) ;  
  
module top;  
reg [0:7] mem[0:7];  
  
initial begin  
mem[0] = 8'b00000111;  
$display("Verilog code says \\"mem[0]=%0b radix b\\\"",mem[0]);  
$display("Verilog code says \\"mem[0]=%0o radix o\\\"",mem[0]);  
$display("Verilog code says \\"mem[0]=%0d radix d\\\"",mem[0]);  
$display("Verilog code says \\"mem[0]=%0h radix h\\\"",mem[0]);  
memcheck_vec(mem);  
end  
  
endmodule
```

The C/C++ function that calls vc_MemoryStringF is as follows:

```
#include <stdio.h>  
#include "DirectC.h"  
  
void memcheck_vec(vc_handle h)  
{  
  
printf("C/C++ code says \\"mem [0] is %s radix b\\\"\n",  
      vc_MemoryStringF(h,0,'b'));  
printf("C/C++ code says \\"mem [0] is %s radix o\\\"\n",  
      vc_MemoryStringF(h,0,'o'));  
printf("C/C++ code says \\"mem [0] is %s radix d\\\"\n",  
      vc_MemoryStringF(h,0,'d'));  
printf("C/C++ code says \\"mem [0] is %s radix x\\\"\n",  
      vc_MemoryStringF(h,0,'x'));  
}
```

The Verilog and C/C++ code display the following:

```
Verilog code says "mem [0]=111 radix b"  
Verilog code says "mem [0]=7 radix o"  
Verilog code says "mem [0]=7 radix d"  
Verilog code says "mem [0]=7 radix h"  
C/C++ code says "mem [0] is 00000111 radix b"  
C/C++ code says "mem [0] is 007 radix o"  
C/C++ code says "mem [0] is 7 radix d"  
C/C++ code says "mem [0] is 07 radix x"
```

void vc_FillWithScalar(vc_handle, scalar)

This routine fills all the bits or a reg, bit, or memory with all 1, 0, x, or z values (you can choose only one of these four values).

You specify the value with the scalar argument, which can be a variable of the scalar type. The scalar type is defined in the DirectC.h file as:

```
typedef unsigned char scalar;
```

You can also specify the value with integer arguments as follows:

0	Specifies 0 values
1	Specifies 1 values
2	Specifies z values
3	Specifies x values

If you declare a scalar type variable, enter it as the argument, and assign only the 0, 1, 2, or 3 integer values to it, they specify filling the Verilog reg, bit, or memory with the 0, 1, z, or x values.

You can use the following definitions from the DirectC.h file to specify these values:

```
#define scalar_0 0
#define scalar_1 1
#define scalar_z 2
#define scalar_x 3
```

The following Verilog and C/C++ code shows you how to use this routine to fill a reg and a memory using the following values:

```
extern void filler (inout reg [7:0] r1,
                    inout reg [7:0] array [1:0] r2,
                    inout reg [7:0] array [1:0] r3);

module top;
reg [7:0] r1;
reg [7:0] r2 [1:0];
reg [7:0] r3 [1:0];
initial
begin
$display("r1 is %0b",r1);
$display("r2[0] is %0b",r2[0]);
$display("r2[1] is %0b",r2[1]);
$display("r3[0] is %0b",r3[0]);
$display("r3[1] is %0b",r3[1]);
filler(r1,r2,r3);
$display("r1 is %0b",r1);
$display("r2[0] is %0b",r2[0]);
$display("r2[1] is %0b",r2[1]);
$display("r3[0] is %0b",r3[0]);
$display("r3[1] is %0b",r3[1]);
end
endmodule
```

The C/C++ code for the function is as follows:

```
#include <stdio.h>
#include "DirectC.h"

filler(vc_handle h1, vc_handle h2, vc_handle h3)
{
    scalar s = 1;
    vc_FillWithScalar(h1,s);
    vc_FillWithScalar(h2,0);
    vc_FillWithScalar(h3,scalar_z);
}
```

The Verilog code displays the following:

```
r1 is xxxxxxxx
r2[0] is xxxxxxxx
r2[1] is xxxxxxxx
r3[0] is xxxxxxxx
r3[1] is xxxxxxxx
r1 is 11111111
r2[0] is 0
r2[1] is 0
r3[0] is zzzzzzzz
r3[1] is zzzzzzzz
```

char *vc_argInfo(vc_handle)

Returns a string containing the information about the argument in the function call in your Verilog source code. For example, if you have the following Verilog source code:

```
extern void show(reg [] array []);
module tester;
reg [31:0] mem [7:0];
reg [31:0] mem2 [16:1];
reg [64:1] mem3 [32:1];
initial begin
    show(mem);
    show(mem2);
    show(mem3);
end
endmodule
```

Verilog memories `mem`, `mem2`, and `mem3` are all arguments to the function named `show`. If that function is defined as follows:

```
#include <stdio.h>
#include "DirectC.h"

void show(vc_handle h)
{
    printf("%s\n", vc_argInfo(h)); /* notice \n after the
string */
}
```

This routine prints the following:

```
input reg[0:31] array[0:7]
input reg[0:31] array[0:15]
input reg[0:63] array[0:31]
```

int vc_Index(vc_handle, U, ...)

Internally, a multi-dimensional array is always stored as a one-dimensional array and this makes a difference in how it can be accessed. In order to avoid duplicating many of the previous access routines for multi-dimensional arrays, the access process is split into two steps. The first step, which this routine performs, is to translate the multiple indices into a single index of a linearized array. The second step is for another access routine to perform an access operation on the linearized array.

This routine returns the index of a linearized array or returns -1 if the U-type parameter is not an index of a multi-dimensional array or the vc_handle parameter is not a handle to a multi-dimensional array of the reg data type.

```
/* get the sum of all elements from a 2-dimensional slice
   of a 4-dimensional array */
int getSlice(vc_handle vh_array, vc_handle vh_idx1,
vc_handle vh_idx2) {

    int sum = 0;
    int i1, i2, i3, i4, idx;

    i1 = vc_getInteger(vh_idx1);
    i2 = vc_getInteger(vh_idx2);
    /* loop over all possible indices for that slice */
    for (i3 = 0; i3 < vc_mdaSize(vh_array, 3); i3++) {

        for (i4 = 0; i4 < vc_mdaSize(vh_array, 4); i4++) {

            idx = vc_Index(vh_array, i1, i2, i3, i4);
            sum += vc_getMemoryInteger(vh_array, idx);
        }
    }
    return sum;
}
```

There are specialized, more efficient versions for two- and three-dimensional arrays. They are as follows:

```
int vc_Index2(vc_handle, U, U)
```

Specialized version of `vc_Index()` where the two U parameters are the indices in a two-dimensional array.

```
int vc_Index3(vc_handle, U, U, U)
```

Specialized version of `vc_Index()` where the two U parameters are the indices in a three-dimensional array.

U vc_mdaSize(vc_handle, U)

Returns the following:

- If the U-type parameter has a value of 0, it returns the number of indices in the multi-dimensional array.
- If the U-type parameter has a value greater than 0, it returns the number of values in the index specified by the parameter. There is an error condition if this parameter is out of the range of indices.
- If the `vc_handle` parameter is not an array, it returns 0.

Summary of Access Routines

[Table 23-6](#) summarizes all the access routines described in the previous section.

Table 23-6 Summary of Access Routines

Access Routine	Description
<code>int vc_isScalar(vc_handle)</code>	Returns a 1 value if the vc_handle is for a one-bit reg or bit. It returns a 0 value for a vector reg or bit or any memory including memories with scalar elements.
<code>int vc_isVector(vc_handle)</code>	This routine returns a 1 value if the vc_handle is to a vector reg or bit. It returns a 0 value for a vector bit or reg or any memory.
<code>int vc_isMemory(vc_handle)</code>	This routine returns a 1 value if the vc_handle is to a memory. It returns a 0 value for a bit or reg that is not a memory.
<code>int vc_is4state(vc_handle)</code>	This routine returns a 1 value if the vc_handle is to a reg or memory that simulates with four states. It returns a 0 value for a bit or a memory that simulates with two states.
<code>int vc_is2state(vc_handle)</code>	This routine does the opposite of the vc_is4state routine.
<code>int vc_is4stVector(vc_handle)</code>	This routine returns a 1 value if the vc_handle is to a vector reg. It returns a 0 value if the vc_handle is to a scalar reg, scalar or vector bit, or to a memory.
<code>int vc_is2stVector(vc_handle)</code>	This routine returns a 1 value if the vc_handle is to a vector bit. It returns a 0 value if the vc_handle is to a scalar bit, scalar or vector reg, or to a memory.
<code>int vc_width(vc_handle)</code>	Returns the width of a vc_handle.
<code>int vc_arraySize(vc_handle)</code>	Returns the number of elements in a memory.
<code>scalar vc_getScalar(vc_handle)</code>	Returns the value of a scalar reg or bit.
<code>void vc_putScalar(vc_handle, scalar)</code>	Passes the value of a scalar reg or bit to a vc_handle by reference.
<code>char vc_toChar(vc_handle)</code>	Returns the 0, 1, x, or z character.
<code>int vc_toInteger(vc_handle)</code>	Returns an int value for a vc_handle to a scalar bit or a vector bit of 32 bits or less.
<code>char *vc_toString(vc_handle)</code>	Returns a string that contains the 1, 0, x, and z characters.

Access Routine	Description
char *vc_toStringF(vc_handle, char)	Returns a string that contains the 1, 0, x, and z characters and allows you to specify the format or radix for the display. The char parameter can be 'b', 'o', 'd', or 'x'.
void vc_putReal(vc_handle, double)	Passes by reference a real (double) value to a vc_handle.
double vc_getReal(vc_handle)	Returns a real (double) value from a vc_handle.
void vc_putValue(vc_handle, char *)	This function passes, by reference through the vc_handle, a value represented as a string containing the 0, 1, x, and z characters.
void vc_putValueF(vc_handle, char, char *)	This function passes by reference through the vc_handle a value for which you specify a radix with the third parameter. The valid radices are 'b', 'o', 'd', and 'x'.
void vc_putPointer(vc_handle, void*) void *vc_getPointer(vc_handle)	These functions pass, by reference to a vc_handle, a generic type of pointer or string. Do not use these functions for passing Verilog data (the values of Verilog signals). Use it for passing C/C++ data. vc_putPointer passes this data by reference to Verilog and vc_getPointer receives this data in a pass by reference from Verilog. You can also use these functions for passing Verilog strings.
void vc_StringToVector(char *, vc_handle)	Converts a C string (a pointer to a sequence of ASCII characters terminated with a null character) into a Verilog string (a vector with 8-bit groups representing characters).
void vc_VectorToString(vc_handle, char *)	Converts a vector value to a string value.
int vc_getInteger(vc_handle)	Same as vc_tolinteger.
void vc_putInteger(vc_handle, int)	Passes an int value by reference through a vc_handle to a scalar reg or bit or a vector bit that is 32 bits or less.

Access Routine	Description
vec32 *vc_4stVectorRef(vc_handle))	Returns a vec32 pointer to a four state vector. Returns NULL if the specified vc_handle is not to a four-state vector reg.
U *vc_2stVectorRef(vc_handle))	This routine returns a U pointer to a bit vector that is larger than 32 bits. If you specify a short bit vector (32 bits or fewer), this routine returns a NULL value.
void vc_get4stVector(vc_handle, vec32 *) void vc_put4stVector(vc_handle, vec32 *)	Passes a four-state vector by reference to a vc_handle to and from an array in C/C++ function. vc_get4stVector receives the vector from Verilog and passes it to the array. vc_put4stVector passes the array to Verilog.
void vc_get2stVector(vc_handle, U *) void vc_put2stVector(vc_handle, U *)	Passes a two state vector by reference to a vc_handle to and from an array in C/C++ function. vc_get2stVector receives the vector from Verilog and passes it to the array. vc_put4stVector passes the array to Verilog.
UB *vc_MemoryRef(vc_handle)	Returns a pointer of type UB that points to a memory in Verilog.
UB *vc_MemoryElemRef(vc_handle, U indx)	Returns a pointer to an element (word, address or index) of a Verilog memory. You specify the vc_handle of the memory and the element.
scalar vc_getMemoryScalar(vc_handle, U indx)	Returns the value of a one-bit memory element.
void vc_putMemoryScalar(vc_handle, U indx, scalar)	Passes a value, of type scalar, to a Verilog memory element. You specify the memory by vc_handle and the element by the indx parameter.
int vc_getMemoryInteger(vc_handle, U indx)	Returns the integer equivalent of the data bits in a memory element whose bit-width is 32 bits or less.
void vc_putMemoryInteger(vc_handle, U indx, int)	Passes an integer value to a memory element that is 32 bits or fewer. You specify the memory by vc_handle and the element by the indx parameter.

Access Routine	Description
void vc_get4stMemoryVector(vc_handle, U indx, vec32 *)	Copies the value in an Verilog memory element to an element in an array. This routine copies both the data and control bytes. It copies them into an array of type vec32.
void vc_put4stMemoryVector(vc_handle, U indx, vec32 *)	Copies Verilog data from a vec32 array to a Verilog memory element.
void vc_get2stMemoryVector(vc_handle, U indx, U *)	Copies the data bytes, but not the control bytes, from a Verilog memory element to an array in your C/C++ function.
void vc_put2stMemoryVector(vc_handle, U indx, U *)	Copies Verilog data from a U array to a Verilog memory element.
void vc_putMemoryValue(vc_handle, U indx, char *)	This routine works like the vc_putValue routine except that it is for passing values to a memory element instead of to a reg or bit. You enter an parameter to specify the element (index) you want the routine to pass the value to.
void vc_putMemoryValueF(vc_handle, U indx, char, char *)	This routine works like the vc_putValueF routine except that it is for passing values to a memory element instead of to a reg or bit. You enter an parameter to specify the element (index) you want the routine to pass the value to.
char *vc_MemoryString(vc_handle, U indx)	This routine works like the vc_toString routine except that it is for passing values to from memory element instead of to a reg or bit. You enter an parameter to specify the element (index) you want the routine to pass the value of.
char *vc_MemoryStringF(vc_handle, U indx, char)	This routine works like the vc_MemoryString function except that you specify a radix with the third parameter. The valid radices are 'b', 'o', 'd', and 'x'.
void vc_FillWithScalar(vc_handle, scalar)	This routine fills all the bits or a reg, bit, or memory with all 1, 0, x, or z values (you can choose only one of these four values).

Access Routine	Description
<code>char *vc_argInfo(vc_handle)</code>	Returns a string containing the information about the parameter in the function call in your Verilog source code.
<code>int vc_Index(vc_handle, U, ...)</code>	Returns the index of a linearized array, or returns -1 if the U-type parameter is not an index of a multi-dimensional array, or the vc_handle parameter is not a handle to a multi-dimensional array of the reg data type.
<code>int vc_Index2(vc_handle, U, U)</code>	Specialized version of vc_Index() where the two U parameters are the indices in a two-dimensional array.
<code>int vc_Index3(vc_handle, U, U, U)</code>	Specialized version of vc_Index() where the two U parameters are the indexes in a three-dimensional array.
<code>U vc_mdaSize(vc_handle, U)</code>	If the U type parameter has a value of 0, it returns the number of indices in multi-dimensional array. If the U type parameter has a value greater than 0, it returns the number of values in the index specified by the parameter. There is an error condition if this parameter is out of the range of indices. If the vc_handle parameter is not a multi-dimensional array, it returns 0.

Enabling C/C++ Functions

The `+vc` compile-time option is required for enabling the direct call of C/C++ functions in your Verilog code. When you use this option you can enter the C/C++ source files on the `vcs` command line. These source files must have a `.c` extension.

There are suffixes that you can append to the `+vc` option to enable additional features. You can append all of them to the `+vc` option in any order. For example:

```
+vc+abstract+allhdrs+list
```

These suffixes specify the following:

+abstract

Specifies that you are using abstract access through vc_handles to the data structures for the Verilog arguments.

When you include this suffix, all functions use abstract access except those with "C" in their declaration; these exceptions use direct access.

If you omit this suffix, all functions use direct access except those with the "A" in their declaration; these exceptions use abstract access.

+allhdrs

Writes the vc_hdtrs.h file that contains external function declarations that you can use in your Verilog code.

+list

Displays on the screen all the functions that you called in your Verilog source code. In this display, void functions are called procedures. The following is an example of this display:

The following external functions have been actually called:

```
procedure receive_string
procedure passbig2
function return_string
procedure passbig1
procedure memory_rewriter
function return_vector_bit
procedure receive_pointer
procedure incr
function return_pointer
function return_reg
```

[DirectC interface]

Mixing Direct And Abstract Access

If you want some C/C++ functions to use direct access and others to use abstract access, you can do so by using a combination of "A" or "C" entries for abstract or direct access in the declaration of the function and the use of the `+abstract` suffix. The following table shows the result of these combinations:

	no <code>+abstract</code> suffix	include the <code>+abstract</code> suffix
extern (no mode specified)	direct access	abstract access
extern "A"	abstract access	abstract access
extern "C"	direct access	direct access

Specifying the DirectC.h File

The C/C++ functions need the `DirectC.h` file in order to use abstract access. This file is located in `$VCS_HOME/include` (and there is a symbolic link to it at `$VCS_HOME/platform/lib/DirectC.h`). You need to tell VCS where to look for it. You can accomplish this in the following three ways:

- Copy the `$VCS_HOME/include/DirectC.h` file to your current directory. VCS will always look for this file in your current directory.
- Establish a link in the current directory to the `$VCS_HOME/include/DirectC.h` file.
- Include the `-CC` option as follows:

```
-CC "-I$VCS_HOME/include"
```

Extended BNF for External Function Declarations

A partial EBNF specification for external function declaration is as follows:

```
source_text ::= description +
description ::= module | user_defined_primitive |
extern_function_declaration
extern_function_declaration ::= extern access_mode
extern_func_type extern_function_name (
list_of_extern_func_args ? ) ;
access_mode ::= ( "A" | "C" ) ?
```

Note:

If access mode is not specified, then the command-line option
+abstract rules; default mode is "C".]

```
extern_func_type ::= void | reg | bit |
```

```

DirectC_primitive_type | bit_vector_type
bit_vector_type ::= bit [ constant_expression :
constant_expression ]
list_of_extern_func_args ::= extern_func_arg
( , extern_func_arg ) *
extern_func_arg ::= arg_direction ? arg_type
optional_arg_name ?

```

Note:

Argument direction (i.e., input, output, inout) applies to all arguments that follow it until the next direction occurs; the default direction is input.

```

arg_direction ::= input | output | inout
arg_type ::= bit_or_reg_type | array_type |
DirectC_primitive_type
bit_or_reg_type ::= ( bit | reg ) optional_vector_range ?
optional_vector_range ::= [ ( constant_expression :
constant_expression ) ? ]
array_type ::= bit_or_reg_type array [ ( constant_expression :
constant_expression ) ? ]
DirectC_primitive_type ::= int | real | pointer | string

```

In this specification, *extern_function_name* and *optional_arg_name* are user-defined identifiers.

24

SAIF Support

The Synopsys Power Compiler enables you to perform power analysis and power optimization for your designs by entering the `power` command at the `vcs` prompt. This command outputs Switching Activity Interchange Format (SAIF) files for your design.

SAIF files support signals and ports for monitoring as well as constructs such as generates, enumerated types, records, array of arrays, and integers.

This chapter covers the following topics:

- [Using SAIF Files](#)
- [SAIF System Tasks](#)
- [Flow to Dump the Backward SAIF File](#)
- [Criteria for Choosing Signals for SAIF Dumping](#)

Using SAIF Files

VCS has native SAIF support so you no longer need to specify any compile-time options to use SAIF files. If you want to switch to the old flow of dumping SAIF files with the PLI, you can continue to give the option `-P $VPOWER_TAB $VPOWER_LIB` to VCS, and the flow will not use the native support.

Note the following when using VCS native support for SAIF files:

- VCS does not need any additional switches.
- VCS does not need a Power Compiler specific tab file (and the corresponding library)
- VCS does not need any additional settings.
- Functionality is built into VCS.

By default VCS does not monitor library cells, the modules in Verilog library files or directories. You can tell VCS to monitor these cells with the `+vcs+saif_libcell` compile-time option.

SAIF System Tasks

This section describes SAIF system tasks that you can use at the command line prompt.

`$set_toggle_region`

Specifies a module instance (or scope) for which VCS records switching activity in the generated SAIF file. Syntax:

`$set_toggle_region(instance[, instance]);`

`$toggle_start`

Instructs VCS to start monitoring switching activity.

Syntax:

```
$toggle_start();
```

`$toggle_stop`

Instructs VCS to stop monitoring switching activity.

Syntax

```
$toggle_stop();
```

`$toggle_reset`

Sets the toggle counter to 0 for all the nets in the current toggle region.

Syntax:

```
$toggle_reset();
```

`$toggle_report`

Reports switching activity to an output file.

Syntax:

```
$toggle_report("outputFile", synthesisTimeUnit,  
Scope);
```

This task has a slight change in native SAIF implementation compared to PLI-based implementation. VCS considers only the arguments specified here for processing. Other arguments have no meaning.

VCS does not report signals in modules defined under the 'celldefine compiler directive.

```
$read_lib_saif
```

Allows you to read in a state dependent and path dependent (SDPD) library forward SAIF file. It registers the state and path dependent information on the scope. It also monitors the internal nets of the design.

Syntax:

```
$read_lib_saif("inputFile");
```

```
$read_rtl_saif
```

Allows you to read in an RTL forward SAIF file. It registers synthesis invariant elements by reading forward SAIF file. By default, it doesn't register internal nets. If neither \$read_lib_saif nor \$read_rtl_saif is specified, then also all the internal nets will be monitored.

Syntax:

```
$read_rtl_saif("inputFile", [, testbench_path_name]);
```

Where:

inputFile

This file enables VCS to register synthesis invariant elements. By default, it doesn't register nets in the output back-annotation SAIF file.

testbench_path_name

The hierarchical name of the instance for which the forwardannotation SAIF file was written. Synopsys recommends that you always include this argument.

```
$set_gate_level_monitoring
```

Allows you to turn on/off the monitoring of nets in the design if both \$read_lib_saif and \$read_rtl_saif are present in the design.

Syntax:

```
$set_gate_level_monitoring("on" | "off" | "rtl_on");  
"rtl_on"  
    All reg type of objects are monitored for toggles. Net  
    type of objects are monitored only if it is a cell highconn. This  
    is the default monitoring policy.
```

```
"off"  
    net type of objects are not monitored.
```

```
"on"  
    reg type of objects are monitored only if it is a cell hiconn.
```

For more details on these task calls, refer to the *Power Compiler User Guide*.

Note:

The \$read_mpm_saif, \$toggle_set, and \$toggle_count tasks in the PLI-based vpower.tab file are obsolete and no longer supported.

Flow to Dump the Backward SAIF File

To generate a backward SAIF file using forward RTL SAIF file, do the following:

```
initial begin  
    $read_rtl_saif(<inputFile>, <Scope>);  
    $set_toggle_region(<Scope>);  
    // initialization of Verilog signals, and then:  
    $toggle_start;  
    // testbench  
    $toggle_stop;  
    $toggle_report(<outputFile>, timeUnit, <Scope>);  
end
```

To generate a backward SAIF file without using the forward RTL SAIF file, do the following:

```
initial begin
    $set_gate_level_monitoring("rtl_on");
    $set_toggle_region(<Scope>);
    // initialization of Verilog signals, and then:
    $toggle_start;
    // testbench
    $toggle_stop;
    $toggle_report(<outputFile>, timeUnit, <Scope>);
end
```

To generate an SDPD backward SAIF file using a forward SAIF file, do the following:

```
initial begin
    $read_lib_saif("inputFile");
    $set_toggle_region(Scope);
    // initialization of Verilog signals, and then:
    $toggle_start;
    // testbench
    $toggle_stop;
    $toggle_report("outputFile", timeUnit, Scope);
end
```

To generate a non-SDPD backward SAIF file without using SAIF files, do the following:

```
initial begin
    $set_gate_level_monitoring("on");
    $set_toggle_region(Scope);
    // initialization of Verilog signals, and then:
    $toggle_start;
    // testbench
    $toggle_stop;
    $toggle_report("outputFile", timeUnit, Scope);
```

```
end
```

Criteria for Choosing Signals for SAIF Dumping

VCS supports only scalar wire and reg, as well as vector wire and reg, for monitoring. It does not consider wire/reg declared within functions, tasks and named blocks for dumping. Also, it does not support bit selects and part selects as arguments to `$set_toggle_region` or `$toggle_report`. In addition, it monitors cell hiconns based on the policy.

25

Encrypting Source Files

You can use VCS to encrypt your HDL source files in such a way that they can be used only with VCS. This chapter describes how to use VCS to encrypt the source files for this purpose.

VCS uses the 128-bit Advanced Encryption Standard (AES) to encrypt the Verilog and VHDL files. The 128-bit key is generated internally by VCS. This 128-bit encryption methodology is exclusive to VCS, and can be decrypted only by VCS.

You can choose to encrypt only certain parts of your source files or entire files using either of the following methods:

- “[Using Compiler Directives or Pragmas](#)” on page 2
- “[Using Automatic Protection Options](#)” on page 5

Using Compiler Directives or Pragmas

You can use VCS to encrypt selected parts of your source files. In order to achieve this, complete the following steps:

1. Enclose the Verilog code that you want to encrypt between the `'protect128` and the `'endprotect128` compiler directives.
Enclose the VHDL code that you want to encrypt between the `--protect128` and `--endprotect128` pragmas.
2. Analyze the files with the `-protect128` option. For example:

```
% vlogan -protect128 foo.v  
% vhdlan -protect128 foo.vhd  
% vcs -protect128 foo.v
```

When you analyze the design with the `-protect128` option, VCS creates new files with the `.vp` or `.vhdp` extension for each Verilog or VHDL file specified at the command line. For example, VCS creates `foo.vp` and `foo.vhdp` when you execute the commands listed above.

In the `.vp` files, VCS replaces the `'protect128` and `'endprotect128` compiler directives with the `'protected128` and `'endprotected128` compiler directives, and encrypts the code in between these directives. In the `.vhdp` files, VCS replaces the `--protect128` and `--endprotect128` pragmas with the `--protected128` and `-endprotected128` pragmas, and encrypts everything in between.

Note:

By default, the encrypted .vp or .vhdp files are saved in the same directory as the source files. You can change this location by using the -putprotect128 analysis option. For example, the following command saves the foo.vp encrypted file in the ./out directory:

```
% vlogan -putprotect128 ./out -protect128 foo.v
```

Note:

- If you specify the protect and protect128 analysis options on the same vcs command line, VCS ignores the protect128 option and uses the protect option. It also reports a warning message.
- The Protect128 and genip options are mutually exclusive, you cannot specify both of these options on the same vcs command line.

Example

The following Verilog file illustrates the use of `protect128 and `endprotect128 to mark the code that needs to be encrypted:

```
module top( inp, outp);
    input [7:0] inp;
    output [7:0] outp;
    reg [7:0] count;
    assign outp = count;
    always
    begin:counter
        `protect128 //begin protected region
        reg [7:0] int;
        count = 0;
        int = inp;
        while (int)
            begin
                if (int [0]) count = count + 1;
```

```

        int = int >> 1;
    end
`endprotect128 //end protected region
end
endmodule

```

The contents of the .vp file that is generated using the -protect128 analysis option is shown below:

```

module top( inp, outp);
    input [7:0] inp;
    output [7:0] outp;
    reg [7:0] count;
    assign outp = count;
    always
    begin:counter
`protected128
P$<-:U="& Y0_+\[?7SYR'AYPDX_H5!KR%>.,^%':>9A_+^UF,6X]=F0S&\-5<;IQ
P:F]/8/)U-%R2 MKD.FB#6?UC"0>XE?R>]^ 3)4@K<.5;*[DX>, +7P@1!S%QA\MME
P>E#R7!*4#IQNK LU):.T[LT=4Y6DP5VWKXN^)F[@L34;C>,=1D'8!9ILX<,AE[6H
P^<P2#1%RY0X??,5)! ,84>FHD @RVX1K=E9UK5,[7Q$^; U\,<JLM#>2@OZ! "!"7
P&ZV60$"CTNE)N+A%]UN19] (H;D,L#V&?&=X) (U!CGVRF3],F!+IC2/KRLG:(-(60
P'>K\BRT_2_/(5^%FBS#-*O$IB[R.;V"1SMJBB:"P4#J="EH".5^?!MYZ#>84>:Q.
`endprotected128
//end protected region
end
endmodule

```

Using Automatic Protection Options

You can encrypt an entire Verilog or VHDL file using the

- autoprotect128, -auto2protect128, or
- auto3protect128 analysis options.

Note:

All these options take precedence over the -protect128 option.
The -auto3protect128 option takes precedence over
-auto2protect128 and -autoprotect128 options,
-auto2protect128 takes precedence over
-autoprotect128, and -autoprotect128 takes precedence
over -protect128.

-autoprotect128

For Verilog files, VCS encrypts the module port list (or UDP terminal list) along with the body of the module (or UDP).

For VHDL files, VCS encrypts the ports, generics, and bodies of entity declarations, and all of the contents of architecture bodies, package declarations, package bodies, and configuration declarations.

-auto2protect128

For Verilog files, VCS encrypts only the body of the module or UDP. It does not encrypt port lists or UDP terminal lists. This option produces a syntactically correct Verilog module or UDP header statement.

For VHDL files, VCS encrypts everything other than the ports in the entity declarations. Though the generated file is syntactically correct file, it may not be semantically correct as the VHDL port declarations can refer to generics in the encrypted portion.

-auto3protect128

This option is similar to the `-auto2protect128` option except for the following differences.

For Verilog files, VCS does not encrypt parameters preceding the ports declaration in a Verilog module.

For VHDL files, VCS does not encrypt the generic clause of entity declarations.

26

Integrating VCS with Vera

Vera® is a comprehensive testbench automation solution for module, block and full system verification. The Vera testbench automation system is based on the OpenVera™ language. This is an intuitive, high-level, object-oriented programming language developed specifically to meet the unique requirements of functional verification.

You can use Vera with VCS to simulate your testbench and design. This chapter describes the required environment settings and usage model to integrate Vera with VCS.

Setting Up Vera and VCS

To use Vera, you must set the Vera environment as shown below:

```
% setenv VERA_HOME Vera_Installation  
% setenv PATH $VERA_HOME/bin:$PATH  
% setenv LM_LICENSE_FILE license_path:$LM_LICENSE_FILE
```

Set the VCS environment as shown below:

```
% setenv VCS_HOME VCS_Installation  
% setenv PATH $VCS_HOME/bin:$PATH  
% setenv LM_LICENSE_FILE license_path:$LM_LICENSE_FILE
```

For more information on VCS installation, see “[Setting Up VCS](#)”.

Using Vera with VCS

The usage model to use Vera with VCS includes the following steps:

- Compile your OpenVera code using Vera

This will generate a `.vro` file and a `filename_vshell.v` file.
The `filename_vshell.v` is a Verilog file.

The following table lists the Vera option to generate a shell file based on your design topology:

Option	Description
<code>-vlog</code>	Generates a Verilog shell file, <code>filename_vshell.v</code> . Use this option if your design is a Verilog-only design.

- Compile your design and the *filename_vshell.v* file using the `-vera` option. This option is required to use Vera with VCS.
- Simulate the design by specifying the `.vro` file created in the first step using the `+vera_load` runtime option. You can also specify this `.vro` file in the `vera.ini` file in your working directory as shown in the following example:

```
vera_load = tb_top.vro
```

See the *Vera User Guide* for more information.

Usage Model

Use the following usage model to compile OpenVera code using Vera:

```
% vera -cmp [Vera_options] OpenVera_files
```

See the *Vera User Guide* for a list of Vera compilation options.

Compilation

```
% vcs [compile_options] -vera verilog_filelist  
filename_vshell.v
```

Simulation

```
% simv [simv_options] +vera_load=file.vro
```


27

Integrating VCS with HSIM

HSIM-VCS DKI is a new mixed-signal simulation implementation using the Synopsys HSIM and VCS simulators. Unlike the VPI/PLI implementation of mixed-signal simulation previously used by HSIM with VPI/PLI standard compliant digital simulators, HSIM-VCS DKI uses a Direct Kernel Interface to exchange information between the HSIM analog simulator and the VCS digital simulator. This approach is similar to the NanoSim-VCS mixed-signal simulation flow and provides more flexibility and better performance over VPI/PLI-based mixed-signal simulation.

HSIM-VCS DKI mixed-signal simulation supports:

- The use of both Verilog top-level netlists and SPICE top-level netlists.
- Donut partitioning, which is the arbitrary instantiation of Spice subcircuits and Verilog modules under either Spice or Verilog throughout the design hierarchy.
- The use of either instance-based or cell-based partitioning.

Environment Setup

A working installation of VCS and a matching version of HSIM are required to run DKI mixed-signal simulation. The compatibility table for versions of HSIM and VCS that work together is available at: <https://solvnet.synopsys.com/retrieve/020828.html>.

The following environment variables must be set:

```
% setenv LM_LICENSE_FILE Location_of_License_File  
or
```

```
% setenv SNPSLMD_LICENSE_FILE license_file_path  
% setenv VCS_HOME VCS_install_directory  
% setenv HSIM_HOME HSIM_install_directory  
% set path = ($VCS_HOME/bin $HSIM_HOME/bin $path)  
% setenv HSIM_64 1
```

Unset the variable `HSIM_64` if you are using in 32-bit mode.

Usage Model

To Compile

The syntax to start the compile process is:

```
% vcs -ad_hsim -ad=init_file Verilog_source_files  
[other_VCS_options]
```

This links the `libvcsdkihsim.so` file and generates an executable called `simv`. A directory called `simv.daidir` is created that contains all of the generated files for VCS and HSIM to run.

`-ad=init_file`

Enables mixed-signal simulation. In the absence of the `init_file`, VCS looks for the default initialization file `vcsAD.init`.

`-ad_hsim`

Enables mixed-signal simulations with HSIM.

To Run Simulation

The syntax to run the mixed-signal simulation is:

```
% simv [runtime_options]
```

This dynamically loads the `libvcsdkihsim.so` file and runs the simulation.

For more information about VCS-HSIM mixed-signal simulation, see the HSIM documentation.

28

Integrating VCS with NanoSim

VCS-NanoSim (VCS-NS) allows a mixed-signal simulation solution, which enables simulating a design that is partly modeled in both analog and digital.

It is recommended that before starting a mixed-signal simulation, both SPICE subcircuits and Verilog modules should be error-free (individually tested).

This chapter briefly describes the environment setup and usage model of VCS-NanoSim mixed-signal simulations. For more information, see the `co_sim.pdf` file in the NanoSim documentation (`/NanoSim_installation/doc/ns/manuals/co_sim.pdf`).

VCS-NanoSim mixed-signal simulation supports:

- The use of both Verilog top-level netlists and SPICE top-level netlists.
- Donut partitioning, which is the arbitrary instantiation of Spice subcircuits and Verilog modules under either Spice or Verilog throughout the design hierarchy.

- The use of either instance-based or cell-based partitioning.
-

Environment Setup

A working installation of VCS and a matching version of NanoSim are required to run mixed-signal simulation. The compatibility table for versions of VCS and NanoSim that work together can be found at: <https://solvnet.synopsys.com/retrieve/020828.html>.

The following environment variables must be set:

Licenses

```
setenv LM_LICENSE_FILE license_file_path
```

or

```
setenv SNPSLMD_LICENSE_FILE license_file_path
```

For NanoSim

```
source NanoSim_install_directory/CSHRC_platform
```

For VCS

```
setenv VCS_HOME vcs_install_directory  
set path = ($VCS_HOME/bin $path)
```

Usage Model

To Compile

The syntax to start the compile process is:

```
% vcs -ad=init_file verilog_source_files [other_vcs_options]  
-ad=init_file
```

Enables mixed-signal simulation. In the absence of the *init_file*, VCS looks for the default initialization file, `vcsAD.init`.

To Run Simulation

The syntax to run the mixed-signal simulation is:

```
% simv [runtime_options]
```


29

Integrating VCS with Specman

The VCS ESI Adapter integrates VCS with the Specman Elite. This chapter describes how to prepare a stand-alone Verilog design for use with the ESI interface. See the *Specman Elite User Guide* for further information.

The VCS ESI adapter is implemented as a Verilog PLI application, and is called `specman.v`. This file is generated using the `specman` command, as explained later in the chapter.

This chapter includes the following topics:

- “Type Support”
- “Usage Flow”
- “Using specrun and specview”
- “Adding Specman Objects To DVE”

Type Support

The VCS ESI adapter supports the following Verilog Types:

- nets
- wires
- registers
- integers
- array of registers (verilog memory)

Other Verilog support:

- Verilog macros
- Verilog tasks
- Verilog functions
- Verilog events
- in/out/inout ports

Usage Flow

This section explains how to integrate Specman with VCS.

Setting Up The Environment

To set up the environment to run Specman with VCS:

- Set your VCS_HOME and VRST_HOME environment variables:

```
% setenv VCS_HOME [vcs_installation_path]
% set path = ($VCS_HOME/bin $path)
% setenv VRST_HOME [specman installation]
```

- Source your env.csh file for Specman:

```
% source ${VRST_HOME}/env.csh
```

For 64-bit simulation, source your env.csh file as shown below:

```
% source ${VRST_HOME}/env.csh -64bit
```

Specman e Code Accessing Verilog

Create the Verilog stub file specman.v and compile all Verilog files including specman.v as shown below:

```
% specman -c "load [top_e_module]; write stubs -verilog;"
```

Compile the design as follows:

- In Compiled mode:

```
% sn_compile.sh -sim vcs \
-sim_flags "[compile-time_options]" specman.v
-debug verilog_filelist" -o simv [top_e_module].e
```

- In Loaded mode:

```
% sn_compile.sh -sim vcs \
-sim_flags "[compile-time_options]" specman.v
-debug verilog_filelist" -o simv
```

Simulate the design as follows:

- In Compiled mode:

```
% simv -ucli [simv_options]
ucli> sn "test"
ucli> run
ucli> quit
```

- In Loaded mode:

```
% simv -ucli [simv_options]
ucli% sn "load <top_e_module>; test"
ucli% run
ucli% quit
```

Note:

Notice the use of the `-o` option with this script to change the name of the executable generated to `simv` from the default name given by the script which is `vcs_specman`.

Using specrun and specview

VCS allows you to use the following Specman utilities to simulate your design:

- `specrun`
- `specview`

`specrun` invokes Specman in batch mode, while `specview` invokes the Specman GUI. The usage model is shown below:

Using specrun

- In Compiled mode:

```
% specrun -p "test -seed=1;" simv [simv_options]
```

- In Loaded mode:

```
% specrun -p "load [top_e_module]; test -seed=1;" \
simv [simv_options]
```

Using specview

Set the environment variable `SPECMAN_OUTPUT_TO_TTY` as shown below:

```
% setenv SPECMAN_OUTPUT_TO_TTY 1
```

- In Compiled mode:

```
% specview -p "test -seed=1;" -sio simv -gui
```

- In Loaded mode:

```
% specview -p "load [top_e_module]; test -seed=1;" \
-sio simv -gui
```

You can also specify VCS runtime options with `specview` or `specrun` as shown in the following examples:

Example 29-1 To Invoke DVE Using specview

The following command invokes the Specman GUI, as well as, DVE.

```
% specview -p "test -seed=1;" -sio simv -gui
```

Similarly, you can also use `-ucli` with `specview` to invoke simulation in UCLI mode.

Example 29-2 To Invoke UCLI Using specrun

The following command invokes the simulation in UCLI mode:

```
% specrun -p "test -seed=1;" simv -ucli -i include.cmd
```

Similarly, you can also use `-gui` with `specrun` to invoke DVE.

Adding Specman Objects To DVE

Following are the steps involved to add e-objects to the DVE wave window:

- Compile the design. See “[Usage Flow](#)”.
- Create the `wave.ecm` file containing the list of e-objects to be added. For example:

```
wave exp sys.U_TbDut.My_Trans  
wave event *.clk
```

- Simulate the design as shown below:

- In Compiled mode:

```
% simv -gui -do run.do
```

Here, the `run.do` contains:

```
sn set wave -mode=manual virsim  
sn config wave -event_data=all_data  
sn test  
sn @wave  
run 8 us
```

- In Loaded mode:

```
% simv -gui -do run.do
```

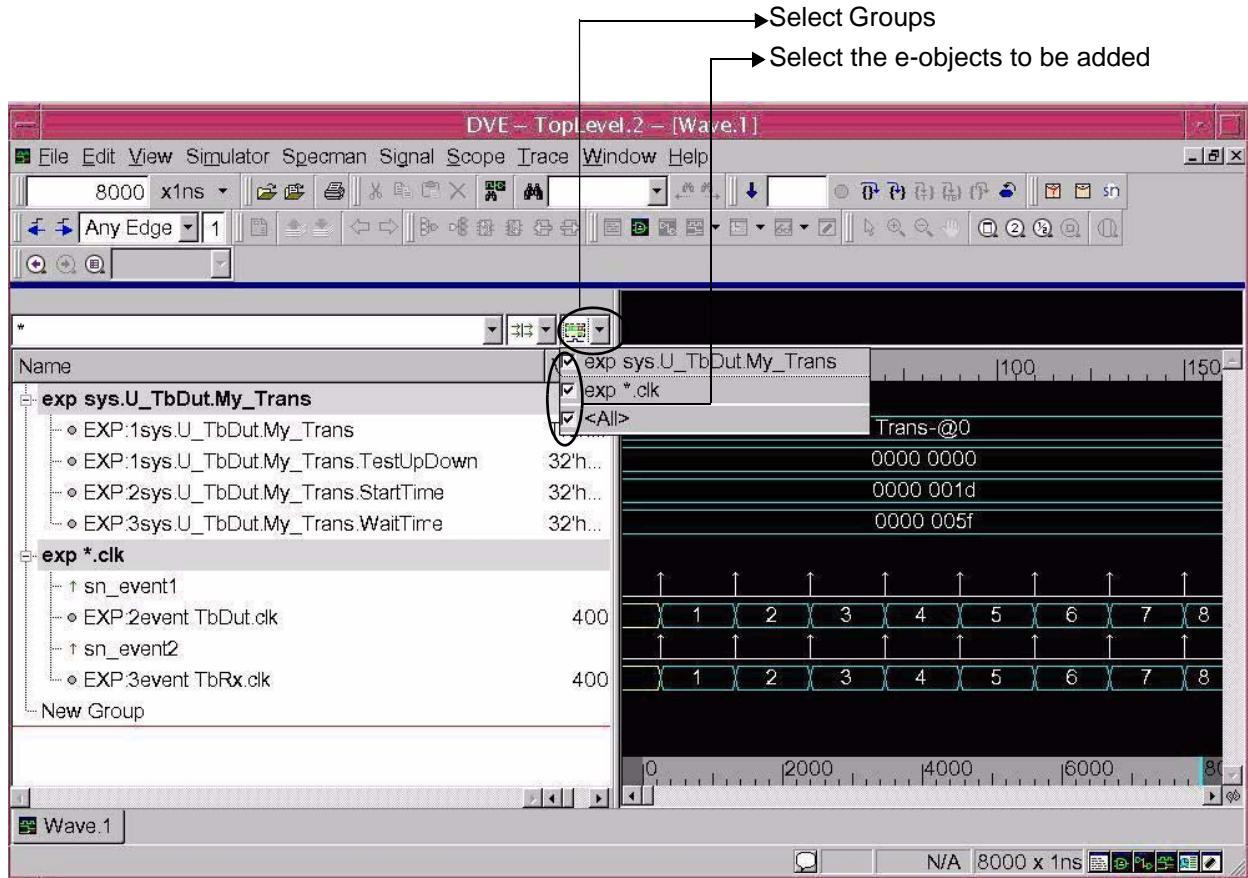
Here, the run.do contains:

```
sn set wave -mode=manual virsim  
sn config wave -event_data=all_data  
sn load top_e_module.e  
sn test  
sn @wave  
run 8 us
```

The simv -gui -do run.do command starts DVE, executes the UCLI commands specified in run.do and creates the sn_wave_sys.cfg configuration file.

- Now, load sn_wave_sys.cfg using **File -> Load Session**, and select sn_wave_sys.cfg.

- Go to the Wave window and click on the groups icon to the side of the filter pane and select the e-objects to be added. See the figure shown below:



30

Integrating VCS with Denali

Denali, a third-party Memory Modeler - Advanced Verification (MMAV) product, can be integrated with VCS through a set of APIs. Denali provides a complete solution for memory modeling and system verification. It automatically monitors all the timing and protocol requirements specified by the memory vendor.

Setting Up Denali Environment for VCS

To use Denali along with VCS, set your Denali environment as shown below:

```
% setenv DENALI [installation_path_of_DENALI]  
% setenv LM_LICENSE_FILE [Denali_license] :$LM_LICENSE_FILE
```

Integrating Denali with VCS

The generic functionality of various memory architectures are captured in a set of highly-optimized 'C' models. The vendor-specific features and the timing for any particular memory device are defined within the specification of memory architecture (SOMA) file. Once the Denali model objects are linked into the simulation environment, modeling any type of memory is as simple as referencing the appropriate SOMA file for that particular memory device.

To access a particular SOMA file, include the following declaration in the source code:

```
parameter memory_spec = soma_file_path;  
parameter init_file = "";
```

Note:

`memory_spec` and `init_file` are keywords.

Usage Model

This section describes the following:

- Usage Model for Verilog Memory Models
- Execute Denali Commands at UCLI Prompt

Usage Model for Verilog Memory Models

Verilog memory models can be integrated with VCS using PLIs. To use Verilog memory models, you need to specify the `pli.tab` file and `denverlib.o` during compilation.

The usage model is shown below:

Compilation

```
% vcs -debug [vcs_options] verilog_filelist \
-P $DENALI/verilog/pli.tab $DENALI/verilog/denverlib.o
```

Note:

To compile the design in 64-bit mode, you must use the
-lpthread option.

Simulation

```
% simv [simv_options]
```

Execute Denali Commands at UCLI Prompt

VCS allows you to execute Denali commands at the UCLI prompt.
For example:

```
% simv -ucli
ucli% mmload :top:I_dut:I_denali_model data_file
```

The above UCLI command loads the Denali memory in the instance
I_denali_model with the data specified in the data_file.

For more information on invoking UCLI, see “[Using Unified Command-line Interface](#)”.

31

Integrating VCS with XA

XA-VCS allows a mixed-signal simulation solution, which enables simulating a design that is partly modeled in both analog and digital.

It is recommended that before starting a mixed-signal simulation, both SPICE subcircuits and Verilog modules should be error-free (individually tested).

This chapter briefly describes the environment setup and usage model of XA-VCS mixed-signal simulations. For more information, please refer Discovery AMS: Mixed-Signal Simulation User Guide in the XA documentation.

XA-VCS mixed-signal simulation supports:

- The use of both Verilog top-level netlists and SPICE top-level netlists.
- Donut partitioning, which is the arbitrary instantiation of SPICE subcircuits and Verilog modules under either SPICE or Verilog throughout the design hierarchy.
- The use of either instance-based or cell-based partitioning.

Environment Setup

A working installation of VCS and a matching version of XA are required to run mixed-signal simulation. The compatibility table for versions of VCS and XA that work together can be found at:
<https://solvnet.synopsys.com/retrieve/020828.html>.

The following environment variables must be set:

Licenses

```
setenv LM_LICENSE_FILE license_file_path
```

or

```
setenv SNPSLMD_LICENSE_FILE license_file_path
```

For XA

```
source XA_install_directory/CSHRC_platform
```

For VCS

```
setenv VCS_HOME vcs_install_directory  
set path = ($VCS_HOME/bin $path)
```

Usage Model

To Compile

The syntax to start the compile process is:

```
% vcs -ad=init_file verilog_source_files  
[other_vcs_options]
```

`-ad=init_file`

Enables mixed-signal simulation. In the absence of the `init_file`, VCS looks for the default initialization file, `vcsAD.init`.

To Run Simulation

The syntax to run the mixed-signal simulation is:

```
% simv [runtime_options]
```


A

VCS Environment Variables

This appendix covers the following topics:

- “Simulation Environment Variables”
- “[Optional Environment Variables](#)”

Simulation Environment Variables

To run VCS, you need to set the following basic environment variables:

\$VCS_HOME

When you or someone at your site installed VCS, the installation created a directory that we call the *vcs_install_dir* directory. Set the \$VCS_HOME environment variable to the path of the *vcs_install_dir* directory. For example:

```
setenv VCS_HOME /u/net/eda_tools/vcs2005.06
```

PATH

On UNIX, set this environment variable to \$VCS_HOME/bin. Add the following directories to your PATH environment variable:

```
set path=($VCS_HOME/bin\  
         $VCS_HOME/'$VCS_HOME/bin/vcs -platform'/bin\  
         $path)
```

Also, make sure the path environment variable is set to a bin directory containing a make or gmake program.

LM_LICENSE_FILE

The definition can either be an absolute path name to a license file or to a port on the license server. Separate the arguments in this definition with colons. For example:

```
setenv LM_LICENSE_FILE 7182@serveroh:/u/net/server/  
eda_tools/license.dat
```

Optional Environment Variables

VCS also includes the following environment variables that you can set in certain circumstances.

DISPLAY_VCS_HOME

Enables the display, at compile time, of the path to the directory specified in the VCS_HOME environment variable. Specify a value other than 0 to enable the display. For example:

```
setenv DISPLAY_VCS_HOME 1
```

PERSISTENT_FLAG

When set to 1, VCS disables the checks enabled by the persistent specification in the tab file. It also disables similar checks that are enabled by the -debug, -debug_all, or -debug_pp options. See the section “[PLI Table File](#)” on page 6.

SYSTEMC_OVERRIDE

Specifies the location of the SystemC simulator used with the VCS/SystemC cosimulation interface. See [Using SystemC](#).

TMPDIR

Specifies the directory used by VCS and the C compiler to store temporary files during compilation.

VCS_CC

Indicates the C compiler to be used. To use the gcc compiler specify the following:

```
setenv VCS_CC gcc
```

VCS_COM

Specifies the path to the VCS compiler executable named vcs1, not the compile script. If you receive a patch for VCS, you might need to set this environment variable to specify the patch. This variable is used for solving problems that require patches from VCS and should not be set by default.

VCS_LIC_EXPIRE_WARNING

By default, VCS displays a warning message 30 days before a license expires. You can specify that this warning message begin fewer days before the license expires with this environment variable, for example:

```
VCS_LIC_EXPIRE_WARNING 5
```

To disable the warning, enter the 0 value:

```
VCS_LIC_EXPIRE_WARNING 0
```

VCS_LOG

Specifies the runtime log file name and location.

VCS_NO_RT_STACK_TRACE

Tells VCS not to return a stack trace when there is a fatal error and instead dump a core file for debugging purposes.

VCS_SWIFT_NOTES

Enables the `printf` PCL command. PCL is the Processor Control Language that works with SWIFT microprocessor models. To enable it, set the value of this environment variable to 1.

B

Compile-time Options

The `vcs` command performs compilation of your design and creates a simulation executable. Compiled event code is generated and used by default. The generated simulation executable, `simv`, can then be used to run multiple simulations.

This section describes the `vcs` command and related options.

Syntax:

```
vcs source_files [source_or_object_files] [options]
```

Here:

source_files

The Verilog or OVA source files for your design separated by spaces.

source_or_object_files

Optional C files (.c), object files (.o), or archived libraries (.a). These are DirectC or PLI applications that you want VCS to link into the binary executable file along with the object files from your Verilog source files. When including object files include the -cc and -ld options to specify the compiler and linker that generated them.

options

Compile-time options that control how VCS compiles your design.

This appendix lists the following:

- “[Options for Accessing Verilog Libraries](#)”
- “[Options for Incremental Compilation](#)”
- “[Options for Help and Documentation](#)”
- “[Options for SystemVerilog Assertions](#)”
- “[Options for Native Testbench](#)”
- “[Options for Different Versions of Verilog](#)”
- “[Options for Initializing Memories and Regs](#)”
- “[Options for Using Radiant Technology](#)”
- “[Options for 64-bit Compilation](#)”

- “Options for Starting Simulation Right After Compilation”
- “Options for Compiling For Coverage Metrics”
- “Options for Debugging Using DVE and UCLI”
- “Options for Specifying Delays and SDF File”
- “Options for Specify Blocks and Timing Checks”
- “Options for Pulse Filtering”
- “Options for Negative Timing Checks”
- “Options for Profiling Your Design”
- “Options to Specify Verilog Source Files and Compile-time Options in a File”
- “Options for Compiling Runtime Options into the Executable”
- “Options for PLI Applications”
- “Options to Enable the VCS DirectC Interface”
- “Options for Flushing Certain Output Text File Buffers”
- “Options for Simulating SWIFT VMC Models and SmartModels”
- “Options for Controlling Messages”
- “Options for Cell Definition”
- “Options for Licensing”
- “Options for Controlling the Linker”
- “Options for Controlling the C Compiler”
- “Options for Source Protection”

- “Options for Mixed Analog/Digital Simulation”
 - “Options for Changing Parameter Values”
 - “Checking for X and Z Values in Conditional Expressions”
 - “Options to Specify the Time Scale”
 - “Options for Overriding Parameters”
 - “General Options”
-

Options for Accessing Verilog Libraries

`-v filename`

Specifies a Verilog library file. VCS looks in this file for definitions of the module and UDP instances that VCS found in your source code, but for which it did not find the corresponding module or UDP definitions in your source code.

`-y directory`

Specifies a Verilog library directory. VCS looks in the source files in this directory for definitions of the module and UDP instances that VCS found in your source code, but for which it did not find the corresponding module or UDP definitions in your source code. VCS looks in this directory for a file with the same name as the module or UDP identifier in the instance (not the instance name). If it finds this file, VCS looks in the file for the module or UDP definition to resolve the instance.

Note:

If you have multiple modules with the same name in different libraries, VCS chooses the module defined in the library that is specified with the first occurrence of the `-y` option.

For example:

If `rev1/cell.v`, `rev2/cell.v` and `rev3/cell.v` all exist and define the module `cell()`, and you issue the following command:

```
% vcs -y rev1 -y rev2 -y rev3 +libext+.v top.v
```

VCS picks `cell.v` from `rev1`.

However, if the `top.v` file has a ``uselib` compiler directive as shown below, then ``uselib` takes priority:

```
//top.v
`uselib directory = /proj/libraries/rev3
//rest of top module code
//end top.v
```

In this case, VCS will use `rev3/cell.v` when you issue the following command:

```
% vcs -y rev1 -y rev2 +libext+.v top.v
```

Include the `+libext` compile-time option to specify the file name extension of the files you want VCS to look for in these directories.

`+incdir+directory+`

Specifies the directory or directories in which VCS searches for include files used in the ``include` compiler directive. More than one directory may be specified, separated by the plus (+) character.

`+libext+extension+`

Specifies that VCS searches only for files with the specified file name extensions in a library directory. You can specify more than one extension, separating the extensions with the plus (+) character. For example, `+libext+.v+.V+` specifies searching for files with either the `.v` or `.V` extension in a library. The order in which you add file name extensions to this option does not specify an order in which VCS searches files in the library with these file name extensions.

`+liborder`

Specifies searching for module definitions for unresolved module instances through the remainder of the library where VCS finds the instance, then searching the next and then the next library on the `vcs` command line before searching in the first library on the command line.

`+librescan`

Specifies always searching libraries for module definitions for unresolved module instances beginning with the first library on the `vcs` command line.

`+libverbose`

Tells VCS to display a message when it finds a module definition in a source file in a Verilog library directory that resolves a module instantiation statement that VCS read in your source files, a library file, or in another file in a library directory. The message is as follows:

Resolving module "*module_identifier*"

By default, VCS does not display this message when it finds a module definition in a Verilog library file that resolves a module instantiation statement.

Options for Incremental Compilation

`-Marchive=number_of_module_definitions`

By default, VCS compiles module definitions into individual object files and sends all the object files in a command line to the linker. Some platforms use a fixed-length buffer for the command line, and if VCS sends too long a list of object files, this buffer overflows and the link fails. A solution to this problem is to have the linker create temporary object files containing more than one module definition so there are fewer object files on the linker command line. With this option, you enable creating these temporary object files and specify how many module definitions are in these files

Using this option briefly doubles the amount of disk space used by the linker because the object files containing more than one module definition are copies of the object files for each module definition. After the linker creates the `simv` executable, it deletes the temporary object files.

-Mdirectory=directory

Specifies the incremental compile directory. The default name for this directory is `csrc`, and its default location is your current directory. You can substitute the shorter `-Mdir` for `-Mdirectory`.

-Mlib=dir

This option provides VCS with a central place to look for the descriptor information before it compiles a module and a central place to get the object files when it links together the executable. This option allows you to use the parts of a design that have been already tested and debugged by other members of your team without recompiling the modules for these parts of the design.

You can specify more than one place for VCS to look for descriptor information and object files by providing multiple arguments with this option.

Example:

```
vcs design.v -Mlib=/design/dir1 -Mlib=/design/dir2
```

Or you can specify more than one directory with this option, using a colon (:) as a delimiter between them, as shown below:

```
vcs design.v -Mlib=/design/dir1:/design/dir2
```

-noIncrComp

Disable incremental compilation.

-parallel_compile_off

Turns off parallel compilation of files and uses serial compilation.

Options for Help and Documentation

`-h` or `-help`

Lists descriptions of the most commonly used VCS compile and runtime options.

`-doc`

Displays the VCS documentation in your system's default web browser.

Options for SystemVerilog Assertions

`-sverilog`

Enables the extensions to the Verilog language specified in the Accellera SystemVerilog specification.

`-ignore keyword_argument`

Suppresses warning messages depending on which keyword argument is specified. The keyword arguments are as follows:

`unique_checks`

Suppresses warning messages about `unique if` and `unique case` statements.

`priority_checks`

Suppresses warning messages about `priority if` and `priority case` statements.

`all`

Suppresses warning messages about unique if, unique case, priority if and priority case statements.

`-assert keyword_argument`

The keyword arguments are as follows:

`enable_diag`

Enables further control of results reporting with runtime options.

`enable_hier`

Enables the use of the runtime option `-assert hier=file.txt`, which allows turning assertions on or off.

`filter_past`

For assertions that are defined with the \$past system task, ignore these assertions when the past history buffer is empty. For instance, at the very beginning of the simulation, the past history buffer is empty. Therefore, the first sampling point and subsequent sampling points should be ignored until the past buffer has been filled with respect to the sampling point.

`disable`

Disables all SystemVerilog assertions in the design.

`disable_cover`

When you include the `-cm assert` compile-time and runtime option, VCS includes information about cover statements in the assertion coverage reports. This keyword prevents cover statements from appearing in these reports.

`disable_file=filename`

Disables the SystemVerilog assertions specified in the file. This file should contain the hierarchical path to your SVA, as shown in the following example:

```
% vcs -assert disable_file=dsbl.txt top  
% cat dsbl.txt  
top.U.SVA1  
top.U.COV1
```

In this example, `dsbl.txt` is disabling the assertions SVA1 and COV1.

`disable_rep_opt`

Specifying a delay or a repetition value greater than 5000 in the assertion expression will affect both compile-time and runtime performance. Therefore, VCS optimizes expression and issues a warning message as shown below:

```
Warning- [LDRF] Large delay or repetition found.  
VCS will optimize compile time. However it may affect runtime.  
Use '-assert disable_rep_opt' to disable this optimization.  
"design.v", 156: (b_ce_idle [* 1:50000])
```

Use `-assert disable_rep_opt` to switch off the optimization and disable this message.

`dumpoff`

Disables the dumping of SVA information in the VPD file during simulation.

`vpiSeqBeginTime`

Enables you to see the simulation time that a SystemVerilog assertion sequence starts when using Debussy.

vpiSeqFail

Enables you to see the simulation time that a SystemVerilog assertion sequence doesn't match when using Debussy.

Options for Native Testbench

`-ntb`

Enables the use of the OpenVera testbench language constructs described in the *OpenVera Language Reference Manual: Native Testbench*.

`-ntb_define macro`

Specifies any OpenVera macro name on the command line. You can specify multiple macro names using the plus (+) character.

The macro can also be defined to be a fixed number. For example, in the following:

```
program test
{
    integer x;
    x =12345;
printf ("DEBUG==> my value = %d and x = %d\n", MYVALUE,
x);
}
```

When you compile and run:

```
% vcs -ntb -ntb_define MYVALUE=10000 myprog.vr -R
```

This outputs are:

```
DEBUG==> my value = 10000 and x = 12345
```

`-ntb_filext .ext`

Specifies an OpenVera file name extension. You can specify multiple file name extensions using the plus (+) character.

`-ntb_incdir directory_path`

Specifies the include directory path for OpenVera files. You can specify multiple include directories using the plus (+) character.

`-ntb_cmp`

Compiles and generates the testbench shell (`file.vshell`) and shared object files.

`-ntb_noshell`

Tells VCS not to generate the shell file. Use this option when you recompile a testbench.

`-ntb_opts keyword_argument`

The keyword arguments are as follows:

`ansi`

Preprocesses the OpenVera files in the ANSI mode. The default preprocessing mode is the Kernighan and Ritchie mode of the C language.

`check`

Does a bounds check on dynamic type arrays (dynamic, associative, queues) and issues an error at runtime.

`check=dynamic`

Same as `check`. Does a bounds check on dynamic type arrays (dynamic, associative, queues) and issues an error at runtime.

`check=fixed`

Does a bounds check only on fixed size arrays and issues an error at runtime.

`check=all`

Does a bounds check on both fixed size and dynamic type arrays and issues an errors at runtime.

`dep_check`

Enables dependency analysis and incremental compilation. Detects files with circular dependencies and issues an error message when VCS cannot determine which file to compile first.

`no_file_by_file_pp`

By default, VCS does file-by-file preprocessing on each input file, feeding the concatenated result to the parser. This argument disables this behavior.

`print_deps`

Tells VCS to display the dependencies for the source files on the screen. Enter this argument with the `dep_check` argument.

`rvm`

Use `rvm` when RVM is used in the testbench.

`tb_timescale=value`

Specifies an overriding timescale for the testbench. The timescale is in the Verilog format (for example, 10ns/10ns).

`tokens`

Preprocesses the OpenVera files to generate two files, `tokens.vr` and `tokens.vrp`. The `tokens.vr` contains the preprocessed result of the non-encrypted OpenVera files, while the `tokens.vrp` contains the preprocessed result of the encrypted OpenVera files. If there is no encrypted OpenVera file, VCS sends all the OpenVera preprocessed results to the `tokens.vr` file.

`use_sigprop`

Enables the signal property access functions. For example, `vera_get_ifc_name()`.

`vera_portname`

Specifies the following:

- The Vera shell module name is named `vera_shell`.
- The interface ports are named `ifc_signal`.
- Bind signals are named, for example, as: `\if_signal [3 : 0]`.

`-ntb_shell_only`

Generates only a `.vshell` file. Use this option when compiling a testbench separately from the design file.

`-ntb_sfname filename`

Specifies the file name of the testbench shell.

`-ntb_sname module_name`

Specifies the name and directory where VCS writes the testbench shell module.

`-ntb_spath`

Specifies the directory where VCS writes the testbench shell and shared object files. The default is the compilation directory.

`-ntb_vipext .ext`

Specifies an OpenVera encrypted-mode file extension to mark files for processing in OpenVera encrypted IP mode. Unlike the `-ntb_filext` option, the default encrypted-mode extensions `.vrp` and `.vrhp` are not overridden and will always be in effect. You can pass multiple file extensions at the same time using the plus (+) character.

`-ntb_vl`

Specifies the compilation of all Verilog files, including the design, the testbench shell file, and the top-level Verilog module.

`+dmprof`

Enables dynamic memory profiler for the testbench.

Options for Different Versions of Verilog

`+systemverilogext+ext`

Specifies a file name extension for SystemVerilog source files. If you use a different file name extension for the SystemVerilog part of your source code and you use this option, you can omit the `-sverilog` option.

`+verilog2001ext+ext`

Specifies a file name extension for Verilog 2001 source files. If you use a different file name extension for the Verilog 2001 part of your source code and you use this option, you can omit the `+v2k` option.

`+verilog1995ext+ext`

Specifies a file name extension for Verilog 1995 files. Using this option allows you to write Verilog 1995 code that would be invalid in Verilog 2001 or SystemVerilog code, such as using Verilog 2001 or SystemVerilog keywords, like `localparam` and `logic`, as names.

Note:

Do not enter all three of these options on the same command line.

Options for Initializing Memories and Regs

`+vcs+initmem+0 | 1 | x | z`

Initializes all bits of all memories in the design.

`+vcs+initreg+0 | 1 | x | z`

Initializes all bits of all regs in the design.

Options for Using Radiant Technology

`+rad`

Performs Radiant Technology optimizations on your design.

`+optconfigfile+filename`

Specifies a configuration file that lists the parts of your design you want to optimize (or not optimize) and the level of optimization for these parts. You can also use the configuration file to specify ACC write capabilities. See “[Compiling With Radiant Technology](#)”.

Options for 64-bit Compilation

`-full64`

Enables compilation and simulation in 64-bit mode.

`-comp64`

Enable 64-bit compilation that generates a 32-bit simv executable.

Options for Starting Simulation Right After Compilation

`-R`

Runs the executable file immediately after VCS links it together.

Options for Compiling For Coverage Metrics

For more detailed information on these options, see the *VCS/VCS MX Coverage Metrics User Guide*.

`-cm line|cond|fsm|tgl|path|branch|assert`

Specifies compiling for the specified type or types of coverage.
The arguments specify the types of coverage:

line

Compile for line or statement coverage.

cond

Compile for condition coverage.

fsm

Compile for FSM coverage.

tgl

Compile for toggle coverage.

path

Compile for path coverage.

branch

Compile for branch coverage

assert

Compile for SystemVerilog assertion coverage.

If you want VCS to compile for more than one type of coverage, use the plus (+) character as a delimiter between arguments, for example:

-cm line+cond+fsm+tgl

The -cm option is also a runtime option and an option on the cmView command line.

`-cm_assert_hier filename`

Limits assertion coverage to the module instances specified in *filename*. Specify the instances using the same format as specified for `-cm_hier` in VCS Coverage Metrics. If this option is not used, coverage is implemented on the whole design.

`-cm_cond arguments`

Modifies condition coverage as specified by the argument or arguments:

`basic`

Only logical conditions and no multiple conditions.

`std`

The default: only logical, multiple, sensitized conditions.

`full`

Logical and non-logical, multiple conditions, no sensitized conditions.

`allops`

Logical and non-logical conditions.

`event`

Signals in event controls in the sensitivity list position are conditions.

`anywidth`

Enables conditions that need more than 32 bits.

sop

Specifies condition SOP coverage.

for

Enables conditions in for loops.

tf

Enables conditions in user-defined tasks and functions.

sop

Tells VCS that when it reads conditional expressions that contain the \wedge bitwise XOR and $\sim \wedge$ bitwise XNOR operators, it should reduce the expression to negation and logical AND or OR.

You can specify more than one argument. You do this by using the plus (+) character between arguments. For example:

-cm_cond basic+allops

-cm_count

Enables cmView to do the following:

- In toggle coverage, reports not just whether a signal toggled from 0 to 1 and 1 to 0, but also the number of times it toggled.
- In FSM coverage, reports not just whether an FSM reached a state, and had such a transition, but also the number of times it did.
- In condition coverage, reports not just whether a condition was met or not, but also the number of times the condition was met.

- In line coverage, reports not just whether a line was executed, but how many times.

-cm_constfile *filename*

Specifies a file listing signals and 0 or 1 values. The format is as shown in the following example:

```
top.t1.a 1
top.t1.b 4 b1010
```

VCS compiles for line and condition coverage as if these signals were permanently at the specified values and you included the **-cm_noconst** option.

-cm_dir *directory_path_name*

Specifies an alternative name and location for the `simv.cm` directory. The **-cm_dir** option is also a runtime option.

Note:

- If you compile the design with the **-cm_dir** option, and then move `simv.cm`, you must use **-cm_dir** at run time to point to the new location of `simv.cm`.
- For generating reports with URG, you need to pass compile time directory first and then rest of the test directories.

-cm_fsmcfg *filename*

Specifies an FSM coverage configuration file. For information on the file format, see the *VCS/VCS MX Coverage Metrics User Guide*.

-cm_fsmopt *keyword_argument*

The keyword arguments are as follows:

`allowTemp`

By default, the variable that holds the current state of the FSM must be directly assigned a numerical constant or the value of a variable that holds the next state of the FSM. This keyword allows FSM extraction when there is indirect assignment to the variable that holds the current state.

`optimist`

Specifies identifying illegal transitions when VCS extracts FSMs in FSM coverage. `cmView` then reports illegal transitions in report files.

`report2StateFsms`

By default, VCS does not extract two state FSMs. This keyword tells VCS to extract them.

`reportvalues`

Specifies reporting the value transitions of the reg that holds the current state of a One Hot or Hot Bit FSM where there are parameters for the bit numbers of the signals that hold the current and next state. The default behavior is to identify these parameters as the states of the FSM and report assignments to their bits as state transitions.

`reportWait`

Enables VCS to monitor transitions when the signal holding the current state is assigned the same state value.

`reportXassign`

Enables the extraction of FSMs in which a state contains the X (unknown) value.

`-cm_fsmresetfilter filename`

Filters out transitions in assignment statements controlled by `if` statements where the conditional expression (following the keyword `if`) is a signal you specify in the file. This filtering out can be for the specified signal in any module definition or in the module definition you specify in the file. You can also specify the FSM and whether the signal is true or false in the file. For information on the file format, see the *VCS/VCS MX Coverage Metrics User Guide*.

`-cm_hier filename`

When compiling for line, condition, toggle or FSM coverage, this option specifies a configuration file that lists the module definitions, instances and sub-hierarchies, and source files that you want VCS to either exclude from coverage or exclusively compile for coverage. For information on the file format, see the *VCS/VCS MX Coverage Metrics User Guide*.

`-cm_ignorepragmas`

Tells VCS to ignore pragmas for coverage metrics.

`-cm_libs yv|celldefine`

Specifies compiling for coverage source files in Verilog libraries when you include the `yv` argument. Specifies compiling for coverage module definitions that are under the `'celldefine` compiler directive when you include the `celldefine` argument. You can specify both arguments together using the plus (+) character.

`-cm_line contassign`

Specifies enabling line coverage for Verilog continuous assignments.

`-cm_name name`

As a compile-time or runtime option, specifies the name of the intermediate data files. When starting cmView, specifies the name of the report files.

`-cm_noconst`

Tells VCS not to monitor for conditions that can never be met or lines that can never execute, because a signal is permanently at a 1 or 0 value. See the *VCS/ VCS MX Coverage Metrics User Guide*.

`-cm_pp [gui] | [batch]`

Tells VCS to start cmView. By default, VCS starts cmView in batch mode.

`gui`

Tells VCS to start the cmView graphical user interface to display coverage data. For information on cmView, see the *VCS/VCS MX Coverage Metrics User Guide*.

`batch`

Tells VCS to start cmView to write reports in batch mode. This is the default operation.

You enter the cmView command line options to the right of this option and its argument.

```
-cm_scope "argument"
```

Limits the scope of what part of the design VCS compiles for coverage. It takes an argument that is similar, but not identical to, a line in the configuration file for the `-cm_hier` option. The difference is that the + (plus) and - (minus) follow the tree, module, file and filelist keywords instead of preceding them as they do in the configuration file. You can enter more than one `-cm_scope` option. The argument must be enclosed in quotation marks. The following is an example of the use of this option:

```
vcs -cm_scope "tree+top.inst1" -cm_scope "file-testsh.v"
```

```
-cm_tgl mda
```

Enables toggle coverage for Verilog-2001 multi-dimensional arrays (MDAs) and SystemVerilog unpacked MDAs. Not required for SystemVerilog packed MDAs.

Options for Debugging Using DVE and UCLI

```
-debug_pp
```

Enables minimal debug access to allow VPD dumping and assertion debugging. You can view the results using DVE in the post-processing mode.

```
-debug
```

Enables interactive debugging using DVE or UCLI. You can force/release values on signals/wires/regs at runtime. It also enables everything that `-debug_pp` offers. However, line stepping is not enabled in this mode.

`-debug_all`

Enables full debugging using DVE or UCLI. You can enable line stepping, force/release any signal/reg/wire, etc. It also enables all other features that `-debug_pp` and `-debug` offer.

`-ucli`

Forces runtime to go into UCLI mode, by default.

`-gui`

When used at compile time, starts DVE at runtime.

`-assert dve`

Enables SystemVerilog assertions tracing in the VPD file. This tracing enables you to see assertion attempts.

`+vpdfile+filename`

Specifies the name of the generated VPD file. You can also use this option for post-processing where it specifies the name of the VPD file.

`+vcdfilename+filename`

Specifies the VCD file you want to use for post-processing.

`+vpdfilesizex+number_in_MB`

Specifies a size for the vpd file. When the vpd file reaches this size, VCS closes this file and opens a new one with the same size.

Options for Specifying Delays and SDF File

`+allmtm`

Specifies compiling separate files for minimum, typical, and maximum delays when there are min:typ:max delay triplets in SDF files. If you use this option, you can use the `+mindelays`, `+typdelays`, or `+maxdelays` options at runtime to specify which compiled SDF file VCS uses. Do not use this option with the `+maxdelays`, `+mindelays`, or `+typdelays` compile-time options.

`+charge_decay`

Enables charge decay in trireg nets. Charge decay will not work if you connect the trireg to a transistor (bidirectional pass) switch such as `tran`, `rtran`, `tranif1`, or `rtranif0`.

`+delay_mode_path`

For modules that contain specify blocks, ignores the delay specifications on all gates and switches and uses only the module path delays and the delay specifications on continuous assignments.

`+delay_mode_zero`

Changes all the delay specifications on all gates, switches, and continuous assignments to zero and changes all module path delays in specify blocks to zero.

`+delay_mode_unit`

Ignores the module path delays in specify blocks and changes all the delay specifications on all gates, switches, and continuous assignments to the shortest time precision argument of all the `'timescale` compiler directives in the source code. The default time unit and time precision argument of the `'timescale` compiler directive is 1s.

`+delay_mode_distributed`

Ignores the module path delays in specify blocks and uses only the delay specifications on all gates, switches, and continuous assignments.

`+maxdelays`

Specifies using the maximum timing delays in min:typ:max delay triplets when compiling the SDF file. The `mtm_spec` argument to the `$sdf_annotate` system task overrides this option.

`+mindelays`

Specifies using the minimum timing delays in min:typ:max delay triplets when compiling the SDF file. The `mtm_spec` argument to the `$sdf_annotate` system task overrides this option.

`+typdelays`

Specifies using the typical timing delays in min:typ:max delay triplets when compiling the SDF file. The `mtm_spec` argument to the `$sdf_annotate` system task overrides this option.

`+multisource_int_delays`

Enables the multisource INTERCONNECT feature, including transport delays with full pulse control.

`+nbaopt`

Removes all intra-assignment delays in all the nonblocking assignment statements in the design. Many users enter a #1 intra-assignment delay in nonblocking procedural assignment statements to make debugging in the Wave window easier. For example:

```
reg1 <= #1 reg2;
```

These delays impede the simulation performance of the design, so after debugging, you can remove these delays with this option.

Note:

The `+nbaopt` option removes all intra-assignment delays in all the nonblocking assignment statements in the design, not just the #1 delays.

`+sdf_nocheck_celltype`

For a module instance to which an SDF file back-annotates delay data, disables comparing the module identifier in the source code with the `CELLTYPE` entry in the SDF file.

`+transport_int_delays`

Enables transport delays for delays on nets with a delay back-annotated from an `INTERCONNECT` entry in an SDF file. The default is inertial delays.

`+transport_path_delays`

Enables transport delays for module path delays.

Options for Specify Blocks and Timing Checks

`+pathpulse`

Enables the search for `PATHPULSE$ specparam` in specify blocks.

`+nospecify`

Suppresses module path delays and timing checks in specify blocks. This option can significantly improve simulation performance.

`+notimingcheck`

Tells VCS to ignore timing check system tasks when it compiles your design. This option can moderately improve simulation performance. The extent of this improvement depends on the number of timing checks that VCS ignores. You can also use this option at runtime to disable these timing checks after VCS has compiled them into the executable. However, the executable simulates faster if you include this option at compile-time so that the timing checks are not in the executable. If you need the delayed versions of the signals in negative timing checks but want faster performance, include this option at runtime. The delayed versions are not available if you use this option at compile-time.

Note:

VCS recognizes `+notimingchecks` to be the same as `+notimingcheck` when you enter it on the vcs or simv command line.

`+no_notifier`

Disables toggling of the notifier register that you specify in some timing check system tasks. This option does not disable the display of warning messages when VCS finds a timing violation that you specified in a timing check.

`+no_tchk_msg`

Disables display of timing violations, but does not disable the toggling of notifier registers in timing checks. This is also a runtime option.

Options for Pulse Filtering

`+pulse_e/number`

Displays an error message and propagates an X value for any path pulse whose width is less than or equal to the percentage of the module path delay specified by the *number* argument, but is still greater than the percentage of the module path delay specified by the *number* argument to the `+pulse_r/number` option.

`+pulse_r/number`

Rejects any pulse whose width is less than *number* percent of the module path delay. The *number* argument is in the range of 0 to 100.

`+pulse_int_r`

Same as the existing `+pulse_r` option, except it applies only to INTERCONNECT delays.

`+pulse_int_e`

Same as the existing `+pulse_e` option, except it applies only to INTERCONNECT delays.

`+pulse_on_event`

Specifies that when VCS encounters a pulse shorter than the module path delay, VCS waits until the module path delay elapses and then drives an X value on the module output port and displays an error message. It drives that X value for a simulation time equal to the length of the short pulse or until another simulation event drives a value on the output port.

`+pulse_on_detect`

Specifies that when VCS encounters a pulse shorter than the module path delay, VCS immediately drives an X value on the module output port, and displays an error message. It does not wait until the module path delay elapses. It drives that X value until the short pulse propagates through the module or until another simulation event drives a value on the output port.

Options for Negative Timing Checks

`-negdelay`

Enables the use of negative values in IOPATH and INTERCONNECT entries in SDF files.

`+neg_tchk`

Enables negative values in timing checks.

`+old_ntc`

Prevents the other timing checks from using delayed versions of the signals in the `$setuphold` and `$recrem` timing checks.

`+NTC2`

In `$setuphold` and `$recrem` timing checks, specifies checking the timestamp and timecheck conditions when the original data and reference signals change value instead of when their delayed versions change value.

`+overlap`

Enables accurate simulation of multiple non-overlapping violation windows for the same signals specified with negative delay values back-annotated from an SDF file to timing checks.

Options for Profiling Your Design

`+prof`

Specifies that VCS writes the `vcs.prof` file during simulation. This file tells you which module definitions, module instances, and Verilog constructs in your design use the most CPU time.

Options to Specify Verilog Source Files and Compile-time Options in a File

`-f filename`

Specify a file that contains a list of Verilog source files (absolute paths) and compile-time options. Note that the following restrictions apply to the contents of this file:

- You can specify all compile-time options that begin with a plus (+) character, except +comp64, +full64, or +memopt.
- You can specify only the following compile-time options that begin with a minus (-) character:

-f	-gen_asm	-gen_obj	
-line	-l	-u	-v
			-y

- You cannot include C source or object files for PLI applications.
- You cannot specify escape characters and meta characters like \$, `, and !.
- You cannot use Verilog comment characters such as // and /* */ to comment out entries in the file.

You can also use the -f option inside this file to point to another file that contains a different list of Verilog source files and compile-time options.

-file *filename*

Specify a file containing a list of files and compile-time options. Use this option to overcome the limitations of the -f option.

Options for Compiling Runtime Options into the Executable

+plusarg_save

Some runtime options must be preceded by the +plusarg_save option for VCS to compile them into the executable.

`+plusarg_ignore`

Tells VCS not to compile the following runtime options into the simv executable. This option is used to counter the `+plusarg_save` option on a previous line.

Options for PLI Applications

`+acc+level_number`

Enables PLI ACC capabilities for the entire design. The level number can be any number between 1 and 4:

`+acc` or `+acc+1`

Enables all capabilities except breakpoints and delay annotation.

`+acc+2`

Above, plus breakpoints.

`+acc+3`

Above, plus module path delay annotation.

`+acc+4`

Above, plus gate delay annotation.

`+applylearn+filename`

Recompiles your design to enable only the ACC capabilities that you needed for the debugging operations you did during a previous simulation of the design.

-e *new_name_for_main*

Specifies the name of your `main()` routine. You write your own `main()` routine when you are writing a C++ application or when your application does some processing before starting the simv executable.

Note:

Do not use the `-e` option with the VCS/SystemC Cosimulation Interface.

-P *pli.tab*

Compiles a user-defined PLI definition table file.

+vpi

Enables the use of VPI PLI access routines.

-load *shared_library:registration_routine*

Specifies the registration routine in a shared library for a VPI application.

-use_vpiobj

Specifies the `vpi_user.c` file that enables you to use the `vpi_register_systf` VPI access routine.

Options to Enable the VCS DirectC Interface

+vc+ [abstract+allhdrs+list]

The `+vc` option enables extern declarations of C/C++ functions and calling these functions in your source code. See the *VCS DirectC Interface User Guide*. The optional suffixes to this option are as follows:

`+abstract`

Enables abstract access through `vc_handles`.

`+allhdrs`

Writes the `vc_hdrs.h` file that contains external function declarations that you can use in your Verilog code.

`+list`

Displays all the C/C++ functions that you called in your Verilog source code.

Options for Flushing Certain Output Text File Buffers

When VCS creates a log, VCD, or text file specified with the `$fopen` system function, VCS writes the data for the file in a buffer and periodically dumps the data from the buffer to the file on disk. The frequency of these dumps varies depending on many factors including the amount of data that VCS has to write to the buffer as simulation or compilation progresses. If you need to see or use the latest information in these files more frequently than the rate at which VCS normally flushes this data, these options tell VCS to flush the data more often during compilation or simulation.

`+vcs+flush+log`

Increases the frequency of flushing both the compilation and simulation log file buffers.

`+vcs+flush+dump`

Increases the frequency of flushing all VCD file buffers.

`+vcs+flush+fopen`

Increases the frequency of flushing all the buffers for the files opened by the `$fopen` system function.

`+vcs+flush+all`

Shortcut option for entering all three of the `+vcs+flush+log`, `+vcs+flush+dump` and `+vcs+flush+fopen` options.

These options do not increase the frequency of dumping other text files, including the VCDE files specified by the `$dumports` system task or the simulation history file for LSI certification specified by the `$lsi_dumports` system task.

These options can also be entered at runtime. Entering them at compile-time modifies the simv executable so that it runs as if these options were always entered at runtime.

Options for Simulating SWIFT VMC Models and SmartModels

`-lmc-swift`

Includes the LMC SWIFT interface.

`-lmc-swift-template`

Generates a Verilog template for a SWIFT Model.

Options for Controlling Messages

`+libverbose`

Tells VCS to display a message when it finds a module definition in a source file in a Verilog library directory that resolves a module instantiation statement that VCS read in your source files, a library file, or in another file in a library directory. The message is:

`Resolving module "module_identifier"`

VCS does not display this message when it finds a module definition in a Verilog library file that resolves a module instantiation statement.

`+lint= [no] ID | none | all`

Enables messages that tell you when your Verilog code contains something that is bad style, but is often used in designs.

`-no_error ID+ID`

Changes the error messages with the UPIMI and IOPCWM IDs to warning messages with the `-no_error` compile-time option. You include one or both IDs as arguments, for example:

`-noerror UPIMI+IOPCWM`

This option does not work with the ID for any other error message.

`-notice`

Enables verbose diagnostic messages.

-q

Quiet mode; suppresses messages such as those about the C compiler VCS is using, the source files VCS is parsing, the top-level modules, or the specified timescale.

-V

Verbose mode; compiles verbosely. The compiler driver program prints the commands it executes as it runs the C compiler, assembler, and linker. If you include the **-R** option with the **-V** option, the **-V** option is also passed to runtime executable, just as if you had entered `simv -V`.

-Vt

Verbose mode; provides CPU time information. Like **-V**, but also prints the amount of time used by each command. Use of the **-Vt** option can cause the simulation to slow down.

+warn= [no] ID | none | all

Uses warning message IDs to enable or disable display of warning messages. In the following warning message:

Warning- [TFIPC] Too few instance port connections

The text string TFIPC is the message ID. The syntax of this option is as follows:

+warn= [no] *ID* | none | all , . . .

Where:

- no Specifies disabling warning messages with the ID that follows. There is no space between the keyword no and the ID.
- none Specifies disabling all warning messages. IDs that follow, in a comma-separated list, specify exceptions.
- all Specifies enabling all warning messages. IDs that follow preceded by the keyword no, in a comma separated list, specify exceptions.

The following are examples that show how to use this option:

+warn=noIPDW	Enables all warning messages except the warning with the IPDW ID.
+warn=none , TFIPC	Disables all warning messages except the warning with the TFIPC ID.
+warn=noIPDW , noTFIPC	Disables the warning messages with the IPDW and TFIPC IDs.
+warn=all	Enables all warning messages. This is the default.

Options for Cell Definition

+nolibcell

Does not define as a cell modules defined in libraries unless they are under the `celldefine compiler directive.

+nocelldefinepli+0

Enables recording in VPD files, the transition times and values of nets and registers in all modules defined under the `celldefine compiler directive or defined in a library that you specify with the -v or -y options. This option also enables full PLI access to these modules.

+nocelldefinepli+1

Disables recording in VPD files, the transition times and values of nets and registers in all modules defined under the `celldefine compiler directive. This option also disables full PLI access to these modules. Modules in a library file or directory are not affected by this option unless they are defined under the `celldefine compiler directive.

+nocelldefinepli+2

In VPD files, disables recording the transition times and values of nets and registers in all modules defined under the `celldefine compiler directive or defined in a library that you specify with the -v or -y options, whether the modules in these libraries are defined under the `celldefine compiler directive or not. This option also disables PLI access to these modules.

Disabling recording of transition times and values of the nets and registers in library cells can significantly increase simulation performance.

Note:

Disabling recording transitions in library cells is intended for batch simulation only and not for interactive debugging with DVE. Any attempt in DVE to access a part of your design for which VPD has been disabled may have unexpected results.

Options for Licensing

+vcs+lic+vcsi

Checks out three VCSI licenses to run VCS.

+vcsi+lic+vcs

Checks out a VCS license to run VCSI when all VCSI licenses are in use.

+vcs+lic+wait

Tells VCS to wait for a network license if none is available.

+vcsi+lic+wait

Tells VCSI to wait for a network license if none is available.

-ID

Returns useful information about a number of things: the version of VCS that you have set the VCS_HOME environment variable to, the name of your work station, your workstation's platform, the host ID of your workstation (used in licensing), the version of the VCS compiler (same as VCS) and the VCS build date.

Options for Controlling the Linker

`-ld linker`

Specifies an alternate front-end linker. Only applicable in incremental compile mode, which is the default.

`-LDFLAGS options`

Passes flag options to the linker. Only applicable in incremental compile mode, which is the default.

`-C`

Tells VCS to compile the source files, generate the intermediate C, assembly, or object files, and compile or assemble the C or assembly code, but not to link them. Use this option if you want to link by hand.

`-lname`

Links the *name* library to the resulting executable. Usage is the letter `l` followed by a name (no space between `l` and *name*). For example: `-lm` (instructs VCS to include the math library).

Options for Controlling the C Compiler

`-cc compiler`

Specifies an alternate C compiler.

`-CC options`

Passes options to the C compiler or assembler.

-CFLAGS *options*

Passes options to C compiler. Multiple **-CFLAGS** are allowed. Allows passing of C compiler optimization levels. For example, if your C code, test.c, calls a library file in your VCS installation under \$VCS_HOME/include, use any of the following **CFLAGS** option arguments:

```
%vcs top.v test.c -CFLAGS "-I$VCS_HOME/include"
```

or

```
%setenv CWD `pwd`  
%vcs top.v test.c -CFLAGS "-I$CWD/include"
```

or

```
%vcs top.v test.c -CFLAGS "-I../include"
```

Note:

The reason to enter ".../include" is because VCS creates a default `csrc` directory where it runs gcc commands. The `csrc` directory is under your current working directory.

Therefore, you need to specify the relative path of the `include` directory to the `csrc` directory for gcc C compiler. Further, you cannot edit files in the `csrc` because VCS automatically creates this directory.

-cpp

Specifies the C++ compiler.

Note:

If you are entering a C++ file or an object file compiled from a C++ file on the vcs command line, you must tell VCS to use the standard C++ library for linking. To do this, enter the -lstdc++ linker flag with the -LDFLAGS elaboration option.

For example:

```
vcs top.v source.cpp -P my.tab \
-cpp /net/local/bin/c++ -LDFLAGS -lstdc++
```

-j number_of_processes

Specifies the number of processes that VCS forks for parallel compilation. There is no space between the "j" character and the number. You can use this option in any compilation mode: directly generating object files from the parallel compilation of your Verilog source files (-gen_obj, default on the Solaris and Linux platforms), generating intermediate assembly files (-gen_asm) and then their parallel assembly, or generating intermediate C files (-gen_c) and their parallel compilation.

-C

Stops after generating the C code intermediate files.

-O0

Suppresses optimization for faster compilation (but slower simulation). Suppresses optimization for how VCS both writes intermediate C code files and compiles these files. This option is the uppercase letter "O" followed by a zero with no space between them.

`-Onumber`

Specifies an optimization level for how VCS both writes and compiles intermediate C code files. The number can be in the 0-4 range; 2 is the default, 0 and 1 decrease optimization, 3 and 4 increase optimization. This option is the uppercase letter "O" followed by 0, 1, 2, 3 or 4 with no space between them. See above, for additional information regarding the `-OO` variant.

`-override-cflags`

Tells VCS not to pass its default options to the C compiler. By default, VCS has a number of C compiler options that it passes to the C compiler. The options it passes depends on the platform, whether it is a 64-bit compilation and other factors. VCS passes these options and then passes the options you specify with the `-CFLAGS` compile-time option.

Options for Source Protection

`+autoprotect [file_suffix]`

Creates a protected source file; all modules are encrypted.

`+auto2protect [file_suffix]`

Creates a protected source file that does not encrypt the port connection list in the module header; all modules are encrypted.

`+auto3protect [file_suffix]`

Creates a protected source file that does not encrypt the port connection list in the module header or any parameter declarations that precede the first port declaration; all modules are encrypted.

`+deleteprotected`

Allows overwriting of existing files when doing source protection.

`+pli_unprotected`

Enables PLI and UCLI access to the modules in the protected source file being created (PLI and UCLI access is normally disabled for protected modules).

`+protect [file_suffix]`

Creates a protected source file, only encrypting ``protect`/
``endprotect` regions.

`+putprotect+target_dir`

Specifies the target directory for protected files.

`+sdfprotect [file_suffix]`

Creates a protected SDF file.

`-Xmangle=number`

Produces a mangled version of input, changing variable names to words from list. Useful to get an entire Verilog design into a single file. Output is saved in the `tokens.v` file. You can substitute `-Xman` for `-Xmangle`.

The argument `number` can be 1, 4, 12, or 28:

`-Xman=1`

Randomly changes names and identifiers, and removes comments, to provide more secure code.

`-Xman=4`

Preserves variable names, but removes comments.

`-Xman=12`

Does the same thing as `-Xman=4`, but also enters, in comments, the original source file name and the line number of each module header.

`-Xman=28`

Does the same thing as `-Xman=12`, but also writes at the bottom of the file comprehensive statistics about the contents of the original source file.

`-Xnomangle=.first | module_identifier, ...`

Specifies module definitions whose module and port identifiers VCS does not change. You use this option with the `-Xman` option. The `.first` argument specifies the module by location (first in file) rather than by identifier. You can substitute `-Xnoman` for `-Xnomangle`.

Options for Mixed Analog/Digital Simulation

`+ad=partition_filename`

Specifies the partition file that you use in mixed Analog/Digital simulation to specify the part of the design simulated by the analog simulator, the analog simulator you want to use, and the resistance mapping information that maps analog drive resistance ranges to Verilog strengths.

`-ams_discipline discipline_name`

Specifies the default discrete discipline in VerilogAMS.

`-ams_iereport`

If information on auto-inserted connect modules (AICMs) is available, displays this information on the screen and in the log file.

`+bidir+1`

Tells VCS to finish compilation when it finds a bidirectional registered mixed-signal net.

`+print+bidir+warn`

Tells VCS to display a list of bidirectional, registered, mixed signal nets.

Options for Changing Parameter Values

`-pvalue+parameter_hierarchical_name=value`

Changes the specified parameter to the specified value.

`-parameters filename`

Changes parameters specified in the file to values specified in the file. The syntax for a line in the file is as follows:

`assign value path_to_parameter`

The path to the parameter is similar to a hierarchical name, except that you use the forward slash character (/) instead of a period as the delimiter.

Checking for X and Z Values in Conditional Expressions

`-xzcheck [nofalseneg]`

Checks all the conditional expressions in the design and displays a warning message every time VCS evaluates a conditional expression to have an X or Z value.

`nofalseneg`

Suppress the warning message when the value of a conditional expression transitions to an X or Z value and then to 0 or 1 in the same simulation time step.

Options to Specify the Time Scale

`-timescale=time_unit/time_precision`

Occasionally, some source files contain the '`'timescale`' compiler directive and others do not. In this case, if you specify the source files that do not contain the '`'timescale`' compiler directive on the command line before you specify the ones that do, this is an error condition and VCS halts compilation, by default. This option enables you to specify the timescale for the source files that do not contain this compiler directive and precede the source files that do. Do not include spaces when specifying the arguments to this option.

`-override_timescale=time_unit/time_precision`

Overrides the time unit and precision unit for all the `timescale compiler directives in the source code, and, similar to the -timescale option, provides a timescale for all module definitions that precede the first `timescale compiler directive. Do not include spaces when specifying the arguments to this option.

Options for Overriding Parameters

-gfile

You can use the `-gfile` compile-time option, to override parameter values through a file.

You need to specify the file name, which contains the list of all parameters that should be overridden, with the `-gfile` option.

The syntax for `-gfile` option is as follows:

```
vcs <top_level_entity_or_module> <other_options>
-gfile <generics_file>
```

The syntax for `generics_file` is as follows:

```
assign <val> <path>
```

Each option In the above syntax is described below:

`val`: The value that overrides the Specified parameter.

`path`: Specifies the absolute hierarchical path to the parameter value which is to be overridden.

Note:

The `-gfile` supports only VHDL syntax for hierarchical path representation.

All escaped identifiers in the Verilog path must be converted into VHDL extended identifiers. If the escaped identifier contains '\' characters, they must be escaped with another '\' character.

For example, consider the following Verilog hierarchical path for the parameter 'P1'.

```
top.dut.\inst1_\cpu .inst2.P1
```

The corresponding `generics_file` entry is as follows:

```
assign 'hffffffff /top/dut/\inst1_\\cpu\\inst2/P1
```

All 'for-generate' and 'instance-array' parentheses must be round parentheses, and the path delimiter must be '/'. All instance paths must start with '/'.

Example:

You can override the parameter and generic values using the `-gfile` option as follows:

```
vcs test.v -gfile overrides.txt
```

where, `overrides.txt` contains the following entries:

```
assign 'hffffffff /top/dut/\inst1_\\cpu\\inst2/P1  
assign "DUMMY" /top/dut/\inst1_\\cpu\\inst2/P2
```

```
assign      10.34 /top/dut/\inst1_\\cpu\inst2/P3
```

Supported Data Types:

The following data types are supported in -gfile option:

- Integer
- Real
- String

The -gfile option ignores other data types with a suitable warning message.

-pvalue

You can use the -pvalue compile-time option for changing the parameter values from the vcs command line.

You specify a parameter with the -pvalue option. It has the following syntax:

```
vcs -pvalue+hierarchical_name_of_parameter=value
```

Example:

```
vcs source.v -pvalue+test.d1.param1=33
```

Note:

The -pvalue option does not work with a localparam or a specparam.

General Options

Enable Verilog 2001 Features

+v2k

Enables language features in the IEEE 1364-2001 standard.

Enable the VCS/SystemC Cosimulation Interface

-sysc

Enables SystemC cosimulation engine.

-sysc=adjust_timeres

Determines the finer time resolution of SystemC and HDL in case of a mismatch, and sets it as the simulator's timescale. VCS may be unable to adjust the time resolution if you elaborate your HDL with the -timescale option or use the sc_set_time_resolution() function call in your SystemC code. In such cases, VCS reports an error and does not create simv.

Note:

You must use this option along with the -sysc option.

TetraMAX

+tetramax

Enables simulation of TetraMAX's testbench in zero delay mode.

Make Accessing an Undeclared Bit an Error Condition

+vcs+boundscheck

Changes reading from, or writing to, an undeclared bit to an error condition instead of a warning condition.

Allow Inout Port Connection Width Mismatches

+noerrorIOPCWM

Changes the error condition, when a signal is wider or narrower than the inout port to which it is connected, to a warning condition, thus allowing VCS to create the simv executable after displaying the warning message.

Allow Zero or Negative Multiconcat Multiplier

-noerror ZONMCM

Changes the following errors to a warning condition, thus allowing VCS to create the simv executable after displaying the warning message:

```
Error- [ZMMCM] Zero multiconcat multiplier cannot be used in this context
A replication with a zero replication constant is considered to have
a size of zero and is ignored. Such a replication shall appear
only within a concatenation in which at least one of the
operands of the concatenation has a positive size.
```

```
target : {0 {1'bx}}
```

```
Error- [NMCM] Negative multiconcat multiplier
target : {(-1) {1'bx}}
"my_test.v", 6
```

VCS errors out if you use "0" or a negative number as a multiconcat multiplier. You can change that error to a warning message using this option.

Specifying a VCD File

`+vcs+dumpvars`

A substitute for entering the `$dumpvars` system task, without arguments, in your Verilog code.

Memories and Multi-Dimensional Arrays (MDAs)

`+memcbk`

Enables callbacks for memories and multi-dimensional arrays (MDAs). Use this option if your design has memories or MDAs and you are doing any of the following:

- Writing a VCD or VPD file during simulation. For VCD files, at runtime, you must also enter the `+vcs+dumparrays` runtime option. For VPD files, you must also enter the `$vcdplusmemon` system task. VCD and VPD files are used for post-processing with DVE.
- Using the VCS/SystemC Interface.
- Writing an FSDB file for Debussy.
- Using any debugging interface application - VCSD/PLI (acc/vpi) that needs to use value change callbacks on memories or MDAs. APIs like `acc_add_callback`, `vcsd_add_callback` and `vpi_register_cb` need this option if these APIs are used on memories or MDAs.

Specifying a Log File

`-l filename`

Specifies a file where VCS records compilation messages. If you also enter the `-R` option, VCS records messages from both compilation and simulation in the same file.

Changing Source File Identifiers to Upper Case

`-u`

Changes all the characters in identifiers to uppercase. It does not change identifiers in quoted strings such as the first argument to the `$monitor` system task. You do not see this change in the DVE Source window, but you do see it in all the other DVE windows.

Defining a Text Macro

`+define+macro=value+`

Defines a text macro in your source code to a value or character string. You can test for this definition in your Verilog source code using the `'ifdef` compiler directive. If there are blank spaces in the character string, then you must enclose it in quotation marks. For example:

```
vcs design.v +define+USELIB="dir=dir1 dir=dir2"
```

The macro is used in a `'uselib` compiler directive:

```
'uselib 'USELIB libext+.v
```

Specifying the Name of the Executable File

-o *name*

Specifies the name of the executable file. In UNIX, the default is simv.

Returning The Platform Directory Name

-platform

Returns the name of the *platform* directory in your VCS installation directory. For example, when you install VCS on a Solaris version 5.4 workstation, VCS creates a directory named, sun_sparc_solaris_5.4, in the directory where you install VCS. In this directory are subdirectories for licensing, executable libraries, utilities, and other important files and executables. You need to set your path to these subdirectories. You can do so by using this option:

```
set path=($VCS_HOME/bin\  
$VCS_HOME/'$VCS_HOME/bin/vcs -platform'/bin\  
$path)
```

Enable Loop Detect

+vcs+loopreport+*number*

Displays a runtime warning message, terminates the simulation, and generates a report when a zero delay loop is detected. By default, VCS checks if a simulation event loops for more than 2,000,000 times during the same simulation time. You can change this default value by specifying any *number* along with this option.

`+vcs+loopdetect+number`

Displays a runtime error message and terminates the simulation when a zero delay loop is detected. By default, VCS checks if a simulation event loops for more than 2,000,000 times during the same simulation time. You can change this default value by specifying any *number* along with this option.

C

Simulation Options

This appendix describes the options and syntax associated with the `simv` executable. These runtime options are typically entered on the `simv` command line but some of them can be compiled into the `simv` executable at compile-time.

This appendix describes the following runtime options:

- “[Options for Simulating Native Testbenches](#)”
- “[Options for SystemVerilog Assertions](#)”
- “[Options for Coverage Metrics](#)”
- “[Options for Enabling and Disabling Specify Blocks](#)”
- “[Options for Specifying When Simulation Stops](#)”
- “[Options for Recording Output](#)”
- “[Options for Controlling Messages](#)”

- “Options for Discovery Visual Environment and UCLI”
 - “Options for VPD Files”
 - “Options for VCD Files”
 - “Options for Specifying Delays”
 - “Options for Flushing Certain Output Text File Buffers”
 - “Options for Licensing”
 - “General Options”
-

Options for Simulating Native Testbenches

-cg_coverage_control

Enables/disables the coverage data collection for all the coverage groups in your NTB-OV or SystemVerilog testbench.

Syntax: -cg_coverage_control=*value*

The valid values for -cg_coverage_control are 0 and 1. A value of 0 disables coverage collection and a value of 1 enables coverage collection.

Note:

You can also use this runtime option with the `coverage_control()` system task. The `coverage_control()` system task enables/disables data collection for one or more coverage groups at the program level. The runtime option takes precedence over the system task. For more information on this system task, refer to the *OpenVera Language Reference Manual: Native Testbench*.

`+ntb_cache_dir`

Specifies the directory location of the cache that VCS maintains as an internal disk cache for randomization.

`+ntb_debug_on_error`

Causes the simulation to stop immediately when it encounters an error. In addition to normal verification errors, this option halts the simulation in case of runtime errors as well.

`+ntb_enable_solver_trace=value`

Enables a debug mode that displays diagnostics when VCS executes a `randomize()` method call. Allowed values are:

0 - Do not display (default).

1 - Displays the constraints VCS is solving.

2 - Displays the entire constraint set.

`+ntb_enable_solver_trace_on_failure [=value]`

Enables a mode that displays trace information only when the VCS constraint solver fails to compute a solution, usually due to inconsistent constraints. When the value of the option is 2, the analysis narrows down to the smallest set of inconsistent constraints, thus aiding the debugging process. Allowed values are 0, 1, and 2. The default value is 2.

`+ntb_exit_on_error [=value]`

Causes VCS to exit when the value is less than 0. The value can be:

0 - continue

1 - exit on first error (default value)

N - exit on nth error

When the value is 0, the simulation finishes regardless of the number of errors.

`+ntb_load=path_name_to_libtb.so`

Specifies loading the testbench shared object file, libtb.so.

`+ntb_random_seed=value`

Sets the seed value to be used by the top-level random number generator at the start of simulation. The random (seed) system function call overrides this setting. The value can be any integer number.

`+ntb_random_seed_automatic`

Picks a unique value to supply as the first seed used by a testbench. The value is determined by combining the time of day, hostname and process id. This ensures that no two simulations have the same starting seed. The

`ntb_random_seed_automatic` seed appears in both the simulation log and the coverage report. When both

`ntb_random_seed_automatic` and `ntb_random_seed` are used, a warning message is printed and the `ntb_random_seed` value is used.

`+ntb_solver_mode=value`

Allows you to choose between one of two constraint solver modes. When set to 1, the solver spends more preprocessing time in analyzing the constraints during the first call to `randomize()` on each class. Therefore, subsequent calls to `randomize()` on that class are very fast. When set to 2, the solver does minimal preprocessing, and analyzes the constraint in each call to `randomize()`. The default is 2.

`+ntb_stop_on_error`

Causes the simulation to stop immediately when it encounters a simulation error, and opens the UCLI debugging environment. In addition to normal verification errors, this option halts the simulation in case of runtime errors. The default setting is to execute the remaining code within the present simulation time.

Options for SystemVerilog Assertions

`-assert keyword_argument`

The keyword arguments are as follows:

`dumpoff`

Disables the dumping of SVA information in the VPD file during simulation.

`finish_maxfail=N`

Terminates the simulation if the number of failures for any assertion reaches `N`. You must supply `N`, otherwise no limit is set.

`global_finish_maxfail=N`

Stops the simulation when the total number of failures, from all SystemVerilog assertions, reaches N .

`maxcover=N`

Disables the collection of coverage information for cover statements after the cover statements are covered N number of times. N must be a positive integer; it cannot be 0.

`maxfail=N`

Limits the number of failures for each assertion to N . When the limit is reached, VCS disables the assertion. You must supply N , otherwise no limit is set.

`maxsuccess=N`

Limits the total number of reported successes to N . You must supply N , otherwise no limit is set. VCS continues to monitor assertions even after the limit is reached.

`nocovdb`

Tells VCS not to write the `program_name.db` database file for assertion coverage.

`nopostproc`

Disables the display of the SVA coverage summary at the end of simulation. A similar summary appears for each `cover` statement:

```
"source_filename.v", line_number:  
cover_statement_hierarchical_name number  
attempts, number total match, number first  
match, number vacuous match
```

`quiet`

Disables the display of messages when assertions fail.

`quiet1`

Disables the display of messages when assertions fail, but enables the display of summary information at the end of simulation. For example:

```
Summary: 2 assertions, 2 with attempts, 2 with  
failures
```

`report [=path/filename]`

Generates a report file in addition to printing results on your screen. By default, the report file name and location is `./assert.report`, but you can change it by entering the `path/filename` argument. The report file name can start with a number or letter. The following special characters are acceptable in the file name: %, ^, and @. Using the following unacceptable special characters: #, &, *, [], \$, (), or ! has the following consequences:

- A file name containing # or & results in a file name truncation to the character before the # or &.
- A file name containing * or [] results in a No match message.
- A file name containing \$ results in an Undefined variable message.
- A file name containing () results in a Badly placed ()'s message.
- A file name containing ! results in an Event not found message.

success

Enables reporting of successful matches, and successes on cover and assert statements respectively, in addition to failures. The default is to report only failures.

vacuous

Enables reporting of vacuous successes on assert statements in addition to the failures. By default, VCS reports only failures.

verbose

Adds more information to the end of the report specified by the report keyword argument, and a summary with the number of assertions present, attempted, and failed.

`hier=file_name`

Specifies a file to enable and disable SystemVerilog assertions when you simulate your design. This feature enables you to control which assertions are active and VCS records in the coverage database, without having to recompile your design.

The types of entries you can make in the file are as follows:

`+tree module_instance_name`

VCS enables the assertions in the specified module instance and all module instances hierarchically under the instance (its child instances).

For example, `+tree top.inst1`. VCS enables the assertions in module instance `top.inst1` and all the assertions in the module instances under this instance.

`-tree module_instance_name`

VCS disables the assertions in the specified module instance and all module instances hierarchically under the instance (its child instances).

For example, `-tree top.inst1.inst2`. VCS disables the assertions in module instance `top.inst1.inst2` and also disables the assertions in the module instances under this instance.

```
+tree assertion_hierarchical_name
```

VCS enables the specified SystemVerilog assertion.

For example, +tree top.inst1.a1. VCS enables the SystemVerilog assertion with the hierarchical name top.inst1.a1.

```
-tree assertion_hierarchical_name
```

VCS disables the specified SystemVerilog assertion.

For example, -tree top.inst1.a2. VCS disables the SystemVerilog assertion with the hierarchical name top.inst1.a2.

```
+module module_identifier
```

VCS enables all the assertions in all instances of the specified module.

For example, +module dev. VCS enables the assertions in all instances of module dev.

```
-module module_identifier
```

VCS disables all the assertions in all instances of the specified module.

For example, -module dev. VCS disables the assertions in all instances of module dev.

```
-assert assertion_block_identifier
```

VCS disables the assertion with the specified block identifier.

You can use wildcard characters in specifying the block identifier to specify more than one assertion.

You can enter more than one keyword using the plus (+) separator. For example:

```
-assert maxfail=10+maxsuccess=20+success+filter.
```

```
-cm assert
```

Specifies monitoring for SystemVerilog assertions coverage.

```
-cm_assert_name path/filename
```

Specifies the path and file name of an initial coverage file. An initial coverage file is needed to set up the database. By default, an empty coverage file is loaded from the following directory:
simv.vdb/snps/fcov.

Options for Coverage Metrics

```
-cm line|cond|fsm|tgl|path|branch|assert
```

Specifies monitoring for the specified type or types of coverage.

The arguments specify the types of coverage:

line

Monitors for line or statement coverage.

cond

Monitors for condition coverage.

`fsm`

Monitors for FSM coverage.

`tgl`

Monitors for toggle coverage.

`path`

Monitors for path coverage.

`branch`

Monitors for branch coverage.

`assert`

Monitors for SystemVerilog assertion coverage.

If you want VCS to monitor for more than one type of coverage, use the plus (+) character as a delimiter between arguments. For example:

```
simv -cm line+cond+fsm+tgl+path
```

The `-cm` option is also a compile-time option and an option on the `cmView` command line.

`-cm_dir directory_path_name`

Specifies an alternative name and location for the `simv.cm` directory. The `-cm_dir` option is also a compile-time option and a `cmView` command-line option.

-cm_glitch *period*

Specifies a glitch period during which VCS does not monitor for coverage caused by value changes. The period is an interval of simulation time specified with a non-negative integer.

-cm_log *filename*

As a compile-time or runtime option, specifies a log file for monitoring for coverage during simulation. As a `cmView` command-line option, specifies a log file for writing reports.

-cm_name *name*

As a compile-time or runtime option, specifies the name of the intermediate data files. On the `cmView` command line, specifies the name of the report files.

-covg_cont_on_error

If the simulation hits an illegal functional coverage bin it will stop. To advance the simulation bypassing this error, enter `-covg_cont_on_error` runtime option with the `simv` command.

+cm_zip

Required when using coverage metrics along with the SWIFT interface to VMC models or SmartModels or when undertow dumping is enabled.

Options for Enabling and Disabling Specify Blocks

`+no_notifier`

Suppresses the toggling of notifier registers that are optional arguments of system timing checks. The reporting of timing check violations is not affected. This is also a compile-time option.

`+no_tchk_msg`

Disables the display of timing violations, but does not disable the toggling of notifier registers in timing checks. This is also a compile-time option.

`+notimingcheck`

Disables timing check system tasks in your design. Using this option at runtime can improve the simulation performance of your design, depending on the number of timing checks that this option disables.

You can also use this option at compile time. Using this option at compile time tells VCS to ignore timing checks when it compiles your design so that the timing checks are not compiled into the executable. This results in a faster simulating executable than one that includes timing checks, which are disabled by this option at runtime.

If you need the delayed versions of the signals in negative timing checks, but want faster performance, include this option at runtime.

Options for Specifying When Simulation Stops

`+vcs+stop+time`

Stop simulation at the *time* value specified. The *time* value must be less than 2^{32} or 4,294,967,296.

`+vcs+finish+time`

Ends simulation at the *time* value specified. The *time* value must be also less than 2^{32} .

For both of these options, there is a special procedure for specifying time values larger than 2^{32} .

Options for Recording Output

`-l filename`

Specifies writing all messages from simulation to the specified file as well as displaying these messages on the standard output.

Options for Controlling Messages

`-q`

Quiet mode; suppresses display of VCS header and summary information. Suppresses the proprietary message at the beginning of simulation and suppresses the VCS Simulation Report at the end (time, CPU time, data structure size, and date).

-V

Verbose mode; displays VCS version and extended summary information. Displays VCS compile and runtime version numbers, and copyright information, at the start of simulation.

+no_pulse_msg

Suppresses pulse error messages, but not the generation of StE values at module path outputs when a pulse error condition occurs.

You can enter this runtime option on the `vcs` command line. You cannot enter this option in the file you use with the `-f` compile-time option.

+sdfverbose

By default, VCS displays no more than ten warning and ten error messages about back-annotating delay information from SDF files. This option enables the display of all back-annotation warning and error messages.

This default limitation on back-annotation messages applies only to messages displayed on the screen and written in the simulation log file. If you specify an SDF log file in the `$sdf_annotation` system task, this log file receives all messages.

+vcs+nostdout

Disables all text output from VCS including messages and text from `$monitor` and `$display` and other system tasks. VCS still writes this output to the log file if you include the `-l` option.

Options for Discovery Visual Environment and UCLI

`-ucli`

Starts the UCLI debugger command line.

`-l log_file_name`

Specifies a log file that contains the commands you entered and the responses from VCS and DVE.

`-do input_file_name`

Specifies a file containing UCLI commands. VCS executes these at the start of simulation.

Options for VPD Files

`+vpdbufsize+number_of_megabytes`

To gain efficiency, VPD uses an internal buffer to store value changes before saving them on disk. This option modifies the size of that internal buffer. The minimum size allowed is what is required to share two value changes per signal. The default size is the size required to store 15 value changes for each signal, but not less than 2 megabytes.

Note:

VCS automatically increases the buffer size as needed to comply with this limit.

`+vpdfile+file_name`

Specifies the name of the output VPD file (default is `vcdplus.vpd`). You must include the full file name with the `.vpd` extension.

`+vpdf filesize +number_of_megabytes`

Creates a VPD file that has a moving window in time while never exceeding the file size specified by *number_of_megabytes*. When the VPD file size limit is reached, VPD continues saving simulation history by overwriting older history.

File size is a direct result of circuit size, circuit activity, and the data being saved. Test cases show that VPD file sizes will likely run from a few megabytes to a few hundred megabytes. Many users can share the same VPD history file, which may be a reason for saving all time value changes when you do simulation. You can save one history file for a design and overwrite it on each subsequent run.

`+vpdf switchsize+number_in_MB`

Specifies a size for the vpd file. When the vpd file reaches this size, VCS closes this file and opens a new one with the same hierarchy as the previous vpd file. There is a number suffix added to all new vpd file names to differentiate them. For example:
`simv +vpdfile+test.vpd +vpdf switchsize+10.`
The first vpd file is named `test.vpd`. When its size reaches 10MB, VCS starts a new file `test_01.vpd`, the third vpd file is `test_02.vpd`, and so on.

`+vpdignore`

Tells VCS to ignore any `$vcdplusxx` system tasks and license checking. By default, VCS checks out a VPD PLI license if there is a `$vcdplusxx` system task in the Verilog source. In some cases, this statement is never executed and VPD PLI license checkout should be suppressed. The `+vpdignore` option performs the license suppression.

`+vpdports`

Causes VPD to store port information, which is then used by the Hierarchy Browser to show whether a signal is a port, and if so, its direction. This option to some extent affects simulation initialization time and memory usage for larger designs.

`+vpdnocompress`

Disables the default compression of data as it is written to the VPD file.

`+vpdnostrengths`

Disables the default storage of strength information on value changes to the VPD file. Use of this option may lead to slight improvements in VCS performance.

Options for VCD Files

`-vcd file_name`

Sets the name of the `$dumpvars` output file to *filename*. The default file name is `verilog.dump`. A `$dumpfile` system task in the Verilog source code overrides this option.

`+vcs+dumpoff+t+ht`

Turns off value change dumping (`$dumpvars`) at time t . ht is the high 32 bits of a time value greater than 32 bits.

`+vcs+dumpon+t+ht`

Suppresses the `$dumpvars` system task until time t . ht is the high 32 bits of a time value greater than 32 bits.

`+vcs+dumparrays`

Enables recording memory and multi-dimensional array values in the VCD file. You must also have used the `+memcbk` compile-time option.

Options for Specifying Delays

`+maxdelays`

Specifies using the maximum delays in min:typ:max delay triplets in module path delays and timing checks, if you compiled your design with the `+allmtm` compile-time option. Also specifies using the maximum timing delays in min:typ:max delay triplets in an uncompiled SDF file.

If you compiled the SDF file with the `+allmtm` compile-time option, the `+maxdelays` option specifies using the compiled SDF file with the maximum delays.

Another use for this runtime option is to specify timing for SWIFT VMC and SmartModels when you also include the `+override_model_delays` runtime option.

`+mindelays`

Specifies using the minimum delays in min:typ:max delay triplets in module path delays and timing checks, if you compiled your design with the `+allmtm` compile-time option. Also specifies using the minimum timing delays in min:typ:max delay triplets in an uncompiled SDF file.

If you compiled the SDF file with the `+allmtm` compile-time option, the `+mindelays` option specifies using the compiled SDF file with the minimum delays.

Another use for this runtime option is to specify timing for SWIFT VMC and SmartModels when you also include the `+override_model_delays` runtime option.

`+typdelays`

Specifies using the typical delays in min:typ:max delay triplets in module path delays and timing checks, if you compiled your design with the `+allmtm` compile-time option. Also specifies using the typical timing delays in min:typ:max delay triplets in an uncompiled SDF file.

If you compiled the SDF file with the `+allmtm` compile-time option, the `+typdelays` option specifies using the compiled SDF file with the typical delays.

This is a default option. By default, VCS uses the typical delay in min:typ:max delay triplets in your source code and in uncompiled SDF files unless you specify otherwise with the *mtm_spec* argument to the `$sdf_annotate` system task. Also, by default, VCS uses the compiled SDF file with typical values.

Another use for this runtime option is to specify timing for SWIFT VMC and SmartModels when you also include the `+override_model_delays` runtime option.

Options for Flushing Certain Output Text File Buffers

When VCS creates a log file, VCD file, or a text file specified with the `$fopen` system function. VCS writes the data for the file in a buffer and periodically dumps the data from the buffer to the file on disk. The frequency of these dumps varies depending on many factors including the amount of data that VCS has to write to the buffer as simulation or compilation progresses. If you need to see or use the latest information in these files more frequently than the rate at which VCS normally dumps this data, these options tell VCS to dump the data more frequently. The amount of frequency also depends on many factors, but the increased frequency will always be significant.

+vcs+flush+log

Increases the frequency of dumping both the compilation and simulation log files.

+vcs+flush+dump

Increases the frequency of dumping all VCD files.

+vcs+flush+fopen

Increases the frequency of dumping all files opened by the `$fopen` system function.

+vcs+flush+all

Increases the frequency of dumping all log files, VCD files, and all files opened by the `$fopen` system function.

These options do not increase the frequency of dumping other text files including the VCDE files specified by the `$dumpports` system task or the simulation history file for LSI certification specified by the `$lsi_dumpports` system task.

You can also enter these options at compile time. There is no performance gain to entering them at compile time.

Options for Licensing

+vcs+lic+vcsi

Checks out three VCSI licenses to run VCS.

+vcsi+lic+vcs

Checks out a VCS license to run VCSI when all VCSI licenses are in use.

+vcs+lic+wait

Waits for a network license if none is available when the job starts.

General Options

Viewing the Compile-Time Options

-E *program*

Starts the *program* that displays the compile-time options that were on the vcs command line when you created the simv (or simv.exe) executable file.

For example: simv -E echo

You cannot use any other runtime options with the -E option.

Recording Where ACC Capabilities are Used

+vcs+learn+pli

ACC capabilities enable debugging operations, but they have a performance cost so you only want to enable them where you need them. This option keeps track of where you use them for debugging operations so that you can recompile your design, and in the next simulation, enable them only where you need them. When you use this option VCS writes the pli_learn.tab secondary PLI table file. You input this file with the +applylearn compile-time option when you recompile your design.

Suppressing the \$stop System Task

`+vcs+ignorestop`

Tells VCS to ignore the \$stop system tasks in your source code.

Enabling User-defined Plusarg Options

`+plus-options`

User-defined runtime options to perform some operation when the option is on the `simv` command line. The `$test$plusargs` system task can check for such options.

Enabling Overriding the Timing of a SWIFT SmartModel

`+override_model_delays`

Instead of using the `DelayRange` parameter definition in the template file, this option enables the `+mindelays`, `+typdelays`, and `+maxdelays` runtime options to specify the timing used by SWIFT SmartModels.

Specifying acc_handle_simulated_net PLI Routine

+vcs+mipd+noalias

For the `acc_handle_simulated_net` PLI routine, aliasing of a loconn net and a hiconn net across the port connection is disabled if MIPD delay annotation happens for the port. If you specify ACC capability: mip or mipb in the `pli.tab` file, such aliasing is disabled only when actual MIPD annotation happens.

If during a simulation run, `acc_handle_simulated_net` is called before MIPD annotation happens, VCS issues a warning message. When this happens you can use this option to disable such aliasing for all ports whenever mip, mipb capabilities have been specified. This option works for reading an ASCII SDF file during simulation and not for compiled SDF files.

D

Compiler Directives and System Tasks

This appendix describes:

- “[Compiler Directives](#)”
- “[System Tasks and Functions](#)”

Compiler Directives

Compiler directives are commands in the source code that specify how VCS compiles the source code that follows them, both in the source files that contain these compiler directives and in the remaining source files that VCS subsequently compiles.

Compiler directives are not effective down the design hierarchy. A compiler directive written above a module definition affects how VCS compiles that module definition, but does not necessarily affect how

VCS compiles module definitions instantiated in that module definition. If VCS has already compiled these lower-level module definitions, it does not recompile them. If VCS has not yet compiled these module definitions, the compiler directive does affect how VCS compiles them.

Note:

Compile-time options override compiler directives.

Compiler Directives for Cell Definition

`celldefine

Specifies that the modules under this compiler directive be tagged as “cell” for delay annotation. See IEEE Std 1364-2001 page 350.

Syntax: `celldefine

`endcelldefine

Disables `celldefine. See IEEE Std 1364-2001 page 350.

Syntax: `endcelldefine

Compiler Directives for Setting Defaults

`default_nettype

Sets default net type for implicit nets. See IEEE Std 1364-2001 page 350.

Syntax: `default_nettype wire | tri | tri0 | wand
| triand | tri1 | wor | trior | trireg | none

``resetall`

Resets all compiler directives. See IEEE 1364-2001 page 357.

Syntax: ``resetall`

Compiler Directives for Macros

``define`

Defines a text macro. See IEEE Std 1364-2001 page 351. Syntax:

``define text_macro_name macro_text`

``else`

Used with ``ifdef`. Specifies an alternative group of source code lines that VCS compiles if the text macro specified with an ``ifdef` compiler directive is not defined. See IEEE Std 1364-2001 page 353. Syntax: ``else second_group_of_lines`

``elseif`

Used with ``ifdef`. Specifies an alternative group of source code lines that VCS compiles if the text macro specified with an `'ifdef` compiler directive is not defined, but the text macro specified with this compiler directive is defined. See IEEE Std 1364-2001 page 353. Syntax: ``elseif text_macro_name second_group_of_lines`

``endif`

Used with ``ifdef`. Specifies the end of a group of lines specified by the ``ifdef` or ``else` compiler directives. See IEEE Std 1364-2001 page 353. Syntax: ``endif`

```
`ifdef
```

Specifies compiling the source lines that follow if the specified text macro is defined by either the `define compiler directive or the +define compile-time option. See IEEE Std 1364-2001 page 353. Syntax: `ifdef *text_macro_name group_of_lines*

The exception is the character string "VCS", which is a predefined text macro in VCS. Therefore, in the following source code, VCS compiles and executes the first block of code and ignores the second block even when you do not include `define VCS or +define+VCS:

```
`ifdef VCS
    begin
        // Block of code for VCS
        .
        .
        .
    end
`else
    begin
        // Alternative block of code
        .
        .
        .
    end
`endif
```

When you encrypt source code, VCS inserts `ifdef VCS before all encrypted parts of the code.

```
`ifndef
```

Specifies compiling the source code that follows if the specified text macro is not defined. See IEEE Std 1364-2001 page 353. Syntax: `ifndef *text_macro_name group_of_lines*

``undef`

Undefines a macro definition. See IEEE Std 1364-2001 page 351.

Syntax: ``undef text_macro_name`

Compiler Directives for Delays

``delay_mode_path`

Ignores the delay specifications on all gates and switches in all those modules under this compiler directive that contain specify blocks. Uses only the module path delays and the delay specifications on continuous assignments. Syntax:

``delay_mode_path`

``delay_mode_distributed`

Ignores the module path delays specified in specify blocks in modules under this compiler directive and uses only the delay specifications on all gates, switches, and continuous assignments. Syntax: ``delay_mode_distributed`

``delay_mode_unit`

Ignores the module path delays. Changes all the delay specifications on all gates, switches, and continuous assignments to the shortest time precision argument of all the `'timescale` compiler directives in the source code. The default time unit and time precision argument of the `'timescale` compiler directive is 1 ns. Syntax: ``delay_mode_unit`

``delay_mode_zero`

Changes all the delay specifications on all gates, switches, and continuous assignments to zero and changes all module path delays to zero. Syntax: ``delay_mode_zero`

Compiler Directives for Backannotating SDF Delay Values

``vcs_mipdexpand`

If the `+oldsdf` compile-time option has been used to turn off SDF compilation at compile-time, this compiler directive enables the runtime back-annotation of individual bits of a port declared in an ASCII text SDF file. This is done by entering the compiler directive over the port declarations for these ports. Similarly, entering this compiler directive over port declarations enables a PLI application to pass delay values to individual bits of a port.

As an alternative to using this compiler directive, you can use the `+vcs+mipdexpand` compile-time option, or you can enter the `mipb ACC` capability. For example:

```
$sdf_annotation call=sdf_annotation_call  
acc+=rw,mipb:top_level_mod+
```

When you compile the SDF file, which Synopsys recommends, you do not need to use this compiler directive to back-annotate the delay values for individual bits of a port.

``vcs_mipdnoexpand`

Turns off the enabling of back-annotating delay values on individual bits of a port as specified by a previous ``vcs_mipdexpand` compiler directive.

Compiler Directives for Source Protection

``endprotect`

Defines the end of code to be protected. Syntax: ``endprotect`

``endprotected`

Defines the end of protected code. Syntax: ``endprotected`

``protect`

Defines the start of code to be protected. Syntax: ``protect`

``protected`

Defines the start of protected code. Syntax: ``protected`

Compiler Directives for Controlling Port Coercion

``noportcoerce`

Does not coerce ports to inout. Syntax: ``noportcoerce`

``portcoerce`

Coerces ports as appropriate (default). Syntax: ``portcoerce`

General Compiler Directives

Compiler Directive for Including a Source File

``include`

Includes source file. See IEEE Std 1364-1995 pages 224-225.

Syntax: ``include "filename"`

Compiler Directive for Setting the Time Scale

``timescale`

Sets the timescale. See IEEE Std 1364-2001 page 357. Syntax:

``timescale time_unit / time_precision`

In VCS, the default time unit is 1 s (a full second) and the default time precision is also 1 s.

Compiler Directive for Specifying a Library

``uselib`

Searches the specified library for unresolved modules. You can specify either a library file or a library directory. Syntax: `'uselib file = filename`

or

``uselib dir = directory_name libext+.ext | libext=.ext`

Enter path names if the library file or directory is not in the current directory. For example:

``uselib file = /sys/project/speclib.lib`

If specifying a library directory, include the `libext+.ext` keyword and append to it the extensions of the source files in the library directory, similar to the `+libext+.ext` compile-time option, for example:

``uselib dir = /net/designlibs/project.lib libext+.v`

To specify more than one search library, enter additional `dir` or `file` keywords, for example:

``uselib dir = /net/designlibs/library1.lib dir=/net/designlibs/library2.lib libext+.v`

Here, the `libext+.ext` keyword applies to both libraries.

Compiler Directive for File Names and Line Numbers

``line line_number "filename" level`

Maintains the file name and line number. See IEEE Std 1364-2001 page 358.

Unimplemented Compiler Directives

The following compiler directives are IEEE Std 1364-1995 compiler directives that are not yet implemented in VCS.

``unconnected_drive`

``nounconnected_drive`

System Tasks and Functions

This section describes the system tasks and functions that are supported by VCS and then lists the system tasks that it does not support.

System tasks that are described in the IEEE Std 1364-2001 are listed with the page number of the description.

System Tasks for SystemVerilog Assertions Severity

`$fatal`

Generates a runtime fatal assertion error. See the *SystemVerilog Language Reference Manual* for VCS/VCS MX, [Chapter 22, "System Tasks and System Functions"](#).

`$error`

Generates a runtime assertion error. See the *SystemVerilog Language Reference Manual for VCS/VCS MX*, [Chapter 22, "System Tasks and System Functions"](#).

`$warning`

Generates a runtime warning message. See the *SystemVerilog Language Reference Manual for VCS/VCS MX*, [Chapter 22, "System Tasks and System Functions"](#).

`$info`

Generates an information message. See the *SystemVerilog Language Reference Manual for VCS/VCS MX*, [Chapter 22, "System Tasks and System Functions"](#).

System Tasks for SystemVerilog Assertions Control

`$assertoff`

Tells VCS to stop monitoring any of the specified assertions that start at a subsequent simulation time. See the *SystemVerilog Language Reference Manual for VCS/VCS MX*, [Chapter 22, "System Tasks and System Functions"](#).

`$assertkill`

Tells VCS to stop monitoring any of the specified assertions that start at a subsequent simulation time, and stop the execution of any of these assertions that are now occurring. See the *SystemVerilog Language Reference Manual for VCS/VCS MX*, [Chapter 22, "System Tasks and System Functions"](#).

`$asserton`

Tells VCS to resume the monitoring of assertions that it stopped monitoring due to a previous `$assertoff` or `$assertkill` system task. See the *SystemVerilog Language Reference Manual for VCS/VCS MX*, Chapter 22, "System Tasks and System Functions".

System Tasks for SystemVerilog Assertions

`$onehot`

Returns true if only one bit in the expression is true. See the *SystemVerilog Language Reference Manual for VCS/VCS MX*, Chapter 22, "System Tasks and System Functions".

`$onehot0`

Returns true if, at the most, one bit of the expression is true (also returns true if none of the bits are true). See the *SystemVerilog Language Reference Manual for VCS/VCS MX*, Chapter 22, "System Tasks and System Functions".

`$isunknown`

Returns true if one of the bits in the expression has an X value. See the *SystemVerilog Language Reference Manual for VCS/VCS MX*, Chapter 22, "System Tasks and System Functions".

System Tasks for VCD Files

VCD files are ASCII files that contain a record of a net or register's transition times and values. There are a number of third-party products that read VCD files to show you simulation results. VCS has the following system tasks for specifying the names and contents of these files:

`$dumpall`

Creates a checkpoint in the VCD file. When VCS executes this system task, VCS writes the current values of all specified nets and registers into the VCD file, whether there is a value change at this time or not. See IEEE std 1364-2001 page 327.

`$dumpoff`

Stops recording value change information in the VCD file. See IEEE std 1364-2001 page 326.

`$dumpon`

Starts recording value change information in the VCD file. See IEEE std 1364-2001 page 326.

`$dumpfile`

Specifies the name of the VCD file you want VCS to record.

Syntax: `$dumpfile ("filename") ;`

`$dumpflush`

Empties the VCD file buffer and writes all this data to the VCD file. See IEEE std 1364-2001 page 328.

`$dumplimit`

Limits the size of a VCD file. See IEEE std 1364-2001 page 327.

`$dumpvars`

Specifies the nets and registers whose transition times and values you want VCS to record in the VCD file. See IEEE std 1364-2001 page 325-326.

Syntax: `$dumpvars (level_number, module_instance | net_or_reg) ;`

You can specify individual nets or registers, or specify all the nets and registers, in an instance.

`$dumpchange`

Tells VCS to stop recording transition times and values in the current dump file and to start recording in the specified new file.

Syntax: `$dumpchange ("filename") ;`

Code example: `$dumpchange ("vcd16a.dmp") ;`

`$fflush`

VCS stores VCD data in the operating system's dump file buffer and as simulation progresses, reads from this buffer to write to the VCD file on disk. If you need the latest information written to the VCD file at a specific time, use the `$fflush` system task.

Syntax: `$fflush("filename") ;`

Code example: `$fflush("vcdfile1.vcd") ;`

`$fflushall`

If you are writing more than one VCD file and need VCS to write the latest information to all these files at a particular time, use the `$fflushall` system task. Syntax: `$fflushall;`

`$gr_waves`

Produces a VCD file with the name `grw.dump`. In this system task, you can specify a display label for a net or register whose transition times and values VCS records in the VCD file. Syntax:
`$gr_waves(["label",] net_or_reg, ...);`

Code example: `$gr_waves("wire w1",w1, "reg r1",r1);`

System Tasks for LSI Certification VCD and EVCD Files

```
$lsi_dumpports
```

For LSI certification of your design, this system task specifies recording a simulation history file that contains the transition times and values of the ports in a module instance. This simulation history file for LSI certification contains more information than the VCD file specified by the \$dumpvars system task. The information in this file includes strength levels and whether the test fixture module (test bench) or the Device Under Test (the specified module instance or DUT) is driving a signal's value.

Syntax:

```
$lsi_dumpports (module_instance, "filename");
```

Code example:

```
$lsi_dumpports (top.middle1, "dumpports.dmp");
```

If you would rather have the \$lsi_dumpports system task generate an extended VCD (EVCD) file instead, include the +dumpports+ieee runtime option.

`$dumpports`

Creates an EVCD file as specified in IEEE Std. 1364-2001 pages 339-340. You can, for example, input a EVCD file into TetraMAX for fault simulation. EVCD files are similar to the simulation history files generated by the `$lsi_dumpports` system task for LSI certification, but there are differences in the internal statements in the file. Further, the EVCD format is a proposed IEEE standard format, whereas the format of the LSI certification file is specified by LSI.

In the past, the `$dumpports` and `$lsi_dumpports` system tasks both generated simulation history files for LSI certification and had identical syntax except for the name of the system task.

Syntax of the `$dumpports` system task is now:

```
$dumpports (module_instance, [module_instance,  
"filename") ;
```

You can specify more than one module instance.

Code example: `$dumpports (top.middle1,top.middle2,
"dumpports.evcd") ;`

If your source code contains a `$dumpports` system task, and you want it to generate simulation history files for LSI certification, include the `+dumpports+lsi` runtime option.

`$dumpportsoff`

Suspends writing to files specified in `$lsi_dumpports` or `$dumpports` system tasks. You can specify a file to which VCS suspends writing or specify no particular file, in which case VCS suspends writing to all files specified by `$lsi_dumpports` or `$dumpports` system tasks. See IEEE Std 1364-2001 page 340-341. Syntax: `$dumpportsoff ("filename") ;`

`$dumpportson`

Resumes writing to the file after writing was suspended by a `$dumpportsoff` system task. You can specify the file to which you want VCS to resume writing or specify no particular file, in which case VCS resumes writing to all files to which writing was halted by any `$dumpportsoff` or `$dumpports` system tasks. See IEEE Std 1364-2001 page 340-341. Syntax:

`$dumpportson ("filename") ;`

`$dumpportsall`

By default, VCS writes to files only when a signal changes value. The `$dumpportsall` system task records the values of the ports in the module instances, which are specified by the `$lsi_dumpports` or `$dumpports` system task, whether there is a value change on these ports or not. You can specify the file to which you want VCS to record the port values for the corresponding module instance or specify no particular file, in which case VCS writes port values in all files opened by the `$lsi_dumpports` or `$dumpports` system task. See IEEE Std 1364-2001 page 341. Syntax:

`$dumpportsall ("filename") ;`

`$dumpportsflush`

VCS stores simulation data in a buffer during simulation from which it writes data to the file. If you want VCS to write all simulation data from the buffer to the file or files at a particular time, execute this `$dumpportsflush` system task. You can specify the file to which you want VCS to write from the buffer or specify no particular file, in which case VCS writes all data from the buffer to all files opened by the `$lsi_dumpports` or `$dumpports` system task. See IEEE Std 1364-2001 page 342.
Syntax: `$dumpportsflush ("filename") ;`

`$dumpportslimit`

Specifies the maximum file size of the file specified by the `$lsi_dumpports` or `$dumpports` system task. You specify the file size in bytes. When the file reaches this limit, VCS no longer writes to the file. You can specify the file whose size you want to limit or specify no particular file, in which case your specified size limit applies to all files opened by the `$lsi_dumpports` or `$dumpports` system task. See IEEE Std 1364-2001 page 341-342.

Syntax: `$dumpportslimit (filesize, "filename") ;`

System Tasks for VPD Files

VPD files are files that store the transition times and values for nets and registers but they differ from VCD files in the following ways:

- You can use the DVE to view the simulation results that VCS recorded in a VPD file. You cannot actually load a VCD file directly into DVE; when you load a VCD file, DVE translates the file to VPD and loads the VPD file.

- They are binary format and therefore take less disk space and load much faster.
- They can also record the order of statement execution so that you can use the Source Window in DVE to step through the execution of your code if you specify recording this information.

VPD files are commonly used in post-processing, where VCS writes the VPD file during batch simulation, and then you review the simulation results using DVE.

There are system tasks that specify the information that VCS writes in the VPD file.

Note:

To use the system tasks for VPD files, you must compile your source code with the `-debug_pp` option.

`$vcdplusautoflushoff`

Turns off the automatic “flushing” of simulation results to the VPD file whenever there is an interrupt, such as when VCS executes the `$stop` system task. Syntax: `$vcdplusautoflushoff;`

`$vcdplusautoflushon`

Tells VCS to “flush” or write all the simulation results in memory to the VPD file whenever there is an interrupt, such as when VCS executes a `$stop` system task or when you halt VCS using the UCLI stop command, or the Stop button on the DVE Interactive window. Syntax: `$vcdplusautoflushon;`

`$vcdplusclose`

Tells VCS to mark the current VPD file as completed, and close the file. Syntax: `$vcdplusclose;`

```
$vcplusplusdeltacycleon
```

Enables delta cycle recording in the VPD file for post-processing.

Syntax: \$vcplusplusevent (*net_or_reg*, "event_name",
"*E|W|I><S|T|D>*");

In DVE, displays a symbol on the signal's waveform and in the Logic Browser. The *event_name* argument appears in the status bar when you click on the symbol.

E|W|I specifies severity. *E* for error, displays a red symbol, *W* for warning, displays a yellow symbol, *I* for information, displays a green symbol.

S|T|D specifies the symbol shape. *S* for square, *T* for triangle, *D* for diamond.

Enter no space between the *E|W|I* and the *S|T|D* arguments. Do not include angle brackets < >.

There is a limit of 244 unique events.

```
$vcplusplusdumpportsoff
```

Tells VCS to suspend writing to VPD file the transition times and values of the module instance specified by

\$vcplusplusdumpports on system task. You can use \$vcplusplusdumpportsoff system task with arguments, but it is not required. Syntax:

```
$vcplusplusdumpportsoff (level_number,  
module_instance) ;
```

```
$vcdplusdumpportson
```

Records transition times and values of ports in a module instance. A level value of 0 tells VCS to dump all levels below the specified instance. If you do not specify a level, the default level is 1. If you use the system task without arguments, VCS dumps all ports of all instances of the while design in the VPD file. Syntax:

```
$vcdplusdumpportson (level_number,  
module_instance) ;
```

Use \$vcdplusdumpportson and \$vcdplusdumpportsoff system tasks to create a VPD file with port drive information for bidirectional ports if you want to use `dumpports` and `dumpvcdports` options in `vpd2vcd` filtering.

Note:

This system task records additional drive information for inout ports of type wire. It does not dump ports with unpacked dimensions. Furthermore, it is unable to determine if a wire is being forced.

```
$vcdplusfile
```

Specifies the next VPD file that DVE opens during simulation, after it executes the `$vcdplusclose` system task and when it executes the next `$vcdpluson` system task. Syntax:

```
$vcdplusfile ("filename") ;
```

```
$vcdplusglitchon
```

Turns on checking for zero delay glitches and other cases of multiple transitions for a signal at the same simulation time.

Syntax: `$vcdplusglitchon`;

`$vcdplusflush`

Tells VCS to “flush” or write all the simulation results in memory to the VPD file at the time VCS executes this system task. Use `$vcdplusautoflushon` to enable automatic flushing of simulation results to the file when simulation stops. Syntax:
`$vcdplusflush;`

`$vcdplusmemon`

Records value changes and times for memories and multi-dimensional arrays. Syntax: `system_task(Mda [, dim1Lsb [, dim1Rsb [, dim2Lsb [, dim2Rsb [, ... dimNLsb [, dimNRsb]]]]])`;

`Mda`

This argument specifies the name of the multi-dimensional array (MDA) to be recorded. It must not be a part select. If no other arguments are given, then all elements of the MDA are recorded to the VPD file.

`dim1Lsb`

This is an optional argument that specifies the name of the variable that contains the left bound of the first dimension. If no other arguments are given, then all elements under this single index of this dimension are recorded.

`dim1Rsb`

This is an optional argument that specifies the name of variable that contains the right bound of the first dimension.

Note:

The `dim1Lsb` and `dim1Rsb` arguments specify the range of the first dimension to be recorded. If no other arguments are given, then all elements under this range of addresses within the first dimension are recorded.

`dim2Lsb`

This is an optional argument with the same functionality as `dim1Lsb`, but refers to the second dimension.

`dim2Rsb`

This is an optional argument with the same functionality as `dim1Rsb`, but refers to the second dimension.

`dimNLsb`

This is an optional argument that specifies the left bound of the *N*th dimension.

`dimNRsb`

This is an optional argument that specifies the right bound of the *N*th dimension.

Note that MDA system tasks can take 0 or more arguments, with the following caveats:

- No arguments: The whole design will be traversed and all memories and MDAs will be recorded. Note that this process may cause significant memory usage and simulator drag.

- One argument: If the object is a scope instance, all memories/MDAs contained in that scope instance and its children will be recorded. If the object is a memory/MDA, that object will be recorded.

`$vcdplusmemoff`

Stops recording value changes and times for memories and multi-dimensional arrays. Syntax is the same as the `$vcdplusmenon` system task.

`$vcdplusmemorydump`

Records (dumps) a snapshot of the values in a memory or multi-dimensional array into the VPD file. Syntax is the same as the `$vcdplusmenon` system task.

`$vcdplusoff`

Stops recording, in the VPD file, the transition times and values for the nets and registers in the specified module instance or individual nets or registers. Syntax:

```
$vcdplusoff [ (level_number, module_instance |  
 net_or_reg) ] ;
```

Where:

level_number

Specifies the number of hierarchy scope levels for which to stop recording signal value changes (a zero value records all scope instances to the end of the hierarchy; default is all).

module_instance

Specifies the name of the scope for which to stop recording signal value changes (default is all).

`net_or_reg`

Specifies the name of the signal for which to stop recording signal value changes (default is all).

`$vcdpluson`

Starts recording, in the VPD file, the transition times and values for the nets and registers in the specified module instance or individual nets or registers. Syntax:

`$vcdpluson [(level_number, module_instance | net_or_variable)] ;`

where:

`level_number`

Specifies the number of hierarchy scope levels for which to record signal value changes (a zero value records all scope instances to the end of the hierarchy; default is all).

`module_instance`

Specifies the name of the scope for which to record signal value changes (default is all).

`net_or_variable`

Specifies the name of the signal for which to record signal value changes (default is all).

`$vcdplustraceoff`

Stops recording, in the VPD file, the order of statement execution in the specified module instance. Syntax:

`$vcdplustraceoff (module_instance) ;`

```
$vcdplustraceon
```

Starts recording, in the VPD file, the order of statement execution in the specified module instance and the module instances hierarchically under it. Syntax:

```
$vcdplustraceon [ (module_instance) ] ;
```

System Tasks for SystemVerilog Assertions

Important:

Enter these system tasks in an initial block. Do not enter them in an always block.

```
$assert_monitor
```

Analogous to the standard \$monitor system task; it continually monitors specified assertions and displays what is happening with them (you can only have it display on the next clock of the assertion). The syntax is as follows:

```
$assert_monitor([0|1,assertion_identifier...]);
```

Where:

0

Specifies reporting on the assertion if it is active (VCS checks for its properties) and if not, reporting on the assertion or assertions, whenever they start.

1

Specifies reporting on the assertion or assertions only once, the next time they start.

If you specify neither 0 or 1, the default is 0.

assertion_identifier...

A comma separated list of assertions. If one of these assertions is not declared in the module definition containing this system task, specify it by its hierarchical name.

`$assert_monitor_off`

Disables the display from the `$assert_monitor` system task.

`$assert_monitor_on`

Re-enables the display from the `$assert_monitor` system task.

System Tasks for Executing Operating System Commands

`$system`

Executes operating system commands. Syntax:

`$system("command");`

Code example: `$system("mv -f savefile savefile.1");`

`$systemf`

Executes operating system commands and accepts multiple formatted string arguments. Syntax: `$systemf ("command %s ...", "string", ...);`

Code example: `int = $systemf ("cp %s %s", "file1", "file2");`

The operating system copies the file named `file1` to a file named `file2`.

System Tasks for Log Files

`$log`

If a filename argument is included, this system task stops writing to the `vcs.log` file or the log file specified with the `-l` runtime option and starts writing to the specified file. If the file name argument is omitted, this system task tells VCS to resume writing to the log file after writing to the file was suspended by the `$nolog` system task. Syntax: `$log [("filename")] ;`

Code example: `$log ("reset.log") ;`

`$nolog`

Disables writing to the `vcs.log` file or the log file specified by either the `-l` runtime option or the `$log` system task. Syntax: `$nolog ;`

System Tasks for Data Type Conversions

`$bitstoreal [b]`

Converts a bit pattern to a real number. See IEEE std 1364-2001 page 310.

`$itor [i]`

Converts integers to real numbers. See IEEE std 1364-2001 page 310.

`$realtobits`

Passes bit patterns across module ports, converting a real number to a 64-bit representation. See IEEE std 1364-2001 page 310.

`$rtoi`

Converts real numbers to integers. See IEEE std 1364-2001 page 310.

System Tasks for Displaying Information

`$display [b|h|0] ;`

Display arguments. See IEEE std 1364-2001 pages 278-285.

`$monitor [b|h|0]`

Display data when arguments change value. See IEEE Std 1364-2001 page 286.

`$monitoroff`

Disables the `$monitor` system task. See IEEE std 1364-2001 page 286.

`$monitoron`

Re-enables the `$monitor` system task after it was disabled with the `$monitoroff` system task. See IEEE std 1364-2001 page 286.

`$strobe [b|h|0] ;`

Displays simulation data at a selected time. See IEEE 1364-2001 page 285.

`$write [b|h|0]`

Displays text. See IEEE std 1364-2001 pages 278-285.

System Tasks for File I/O

`$fclose`

Closes a file. See IEEE std 1364-2001 pages 286-288.

`$fdisplay [b|h|0]`

Writes to a file. See IEEE std 1364-2001 pages 288-289.

`$ferror`

Returns additional information about an error condition in file I/O operations. See IEEE Std 1364-2001 pages 294-295.

`$fflush`

Writes buffered data to files. See IEEE Std 1364-2001 page 294.

`$fgetc`

Reads a character from a file. See IEEE Std 1364-2001 page 290.

`$fgets`

Reads a string from a file. See IEEE Std 1364-2001 page 290.

`$fmonitor [b|h|0]`

Writes to a file when an argument changes value. See IEEE std 1364-2001 pages 287-288.

`$fopen`

Opens files. See IEEE std 1364-2001 pages 286-288.

`$fread`

Reads binary data from a file. See IEEE Std 1364-2001 page 293.

`$fscanf`

Reads characters in a file. See IEEE Std 1364-2001 pages 290-293.

`$fseek`

Sets the position of the next read or write operation in a file. See IEEE Std 1364-2001 page 294.

`$fstrobe [b | h | 0]`

Writes arguments to a file. See IEEE std 1364-2001 pages 288-289.

`$ftell`

Returns the offset of a file. See IEEE Std 1364-2001 page 294.

`$fwrite [b | h | 0]`

Writes to a file. See IEEE Std 1364-2001 pages 88-289.

`$rewind`

Sets the next read or write operation to the beginning of a file. See IEEE Std 1364-2001 page 294.

`$sformat`

Assigns a string value to a specified signal. See IEEE Std 1364-2001 pages 289-290.

`$sscanf`

Reads characters from an input stream. See IEEE Std 1364-2001 pages 290-293.

`$swrite`

Assigns a string value to a specified signal, similar to the `$sformat` system function. See IEEE Std 1364-2001 pages 289-290.

`$ungetc`

Returns a character to the input stream. See IEEE Std 1364-2001 page 290.

System Tasks for Loading Memories

`$readmemb`

Loads binary values in a file into memories. See IEEE std 1364-2001 pages 295-296.

`$readmemh`

Loads hexadecimal values in a file into memories. See IEEE std 1364-2001 pages 295-296.

`$sreadmemb`

Loads specified binary string values into memories. See IEEE std 1364-2001 page 744.

`$sreadmemh`

Loads specified string hexadecimal values into memories. See IEEE std 1364-2001 page 744.

`$writememb`

Writes binary data in a memory to a file. Syntax: `$writememb ("filename", memory [, start_address] [, end_address]);`

Code example: `$writememb ("testfile.txt", mem, 0, 255);`

`$writememh`

Writes hexadecimal data in a memory to a file. Syntax:
`$writememh ("filename", memory [, start_address] [, end_address]);`

System Tasks for Time Scale

`$printtimescale`

Displays the time unit and time precision from the last
'timescale compiler directive that VCS has read before it reads
the module definition containing this system task. See IEEE std
1364-2001 pages 297-298.

`$timeformat`

Specifies how the %t format specification reports time
information. See IEEE std 1364-2001 pages 298-301.

System Tasks for Simulation Control

`$stop`

Halts simulation. See IEEE std 1364-2001 pages 301-302.

```
$finish
```

Ends simulation. See IEEE std 1364-2001 page 301.

System Tasks for Timing Checks

```
$disable_warnings
```

Disables the display of timing violations but does not disable the toggling of notifier registers. Syntax:

```
$disable_warnings [(module_instance, . . .)] ;
```

An alternative syntax is:

```
$disable_warnings ("timing" [, module_instance, . . .]) ;
```

If you specify a module instance, this system task disables timing violations for the specified instance and all instances hierarchically under this instance. If you omit module instances, this system task disables timing violations throughout the design.

Code example: `$disable_warnings (seqdev1) ;`

`$enable_warnings`

Re-enables the display of timing violations after the execution of the `$disable_warnings` system task. This system task does not enable timing violations during simulation when you used the `+no_tchk_msg` compile-time option to disable them. Syntax:
`$enable_warnings [(module_instance, . . .)] ;`

An alternative syntax is:

`$enable_warnings("timing" [,module_instance, . . .]) ;`

If you specify a module instance, this system task enables timing violations for the specified instance and all instances hierarchically under this instance. If you omit module instances, this system task enables timing violations throughout the design.

`$hold`

Reports a timing violation when a data event happens too soon after a reference event. See IEEE Std 1364-2001 pages 241-242.

`$period`

Reports a timing violation when an edge triggered event happens too soon after the previous matching edge triggered an event on a signal. See IEEE Std 1364-2001 pages 255-256.

`$recovery`

Reports a timing violation when a data event happens too soon after a reference event. Unlike the `$setup` timing check, the reference event must include the `posedge` or `negedge` keyword. Typically the `$recovery` timing check has a control signal, such as `clear`, as the reference event, and the clock signal as the data event. See IEEE 1364-2001 pages 245-246.

`$recrem`

Reports a timing violation if a data event occurs less than a specified time limit before or after a reference event. This timing check is identical to the `$setuphold` timing check except that typically the reference event is on a control signal and the data event is on a clock signal. You can specify negative values for the recovery and removal limits. The syntax is as follows:

```
$recrem(reference_event, data_event,  
recovery_limit, removal_limit, notifier,  
timestamp_cond, timecheck_cond, delay_reference,  
delay_data);
```

See IEEE Std 1364-2001 pages 246-248.

`$removal`

Reports a timing violation if the reference event, typically an asynchronous control signal, happens too soon after the data event, the clock signal. See IEEE Std 1364-2001 pages 244-245.

`$setup`

Reports a timing violation when the data event happens before and too close to the reference event. See IEEE Std 1364-2001 page 241. This timing check also has an extended syntax like the `$recrem` timing check. This extended syntax is not described in IEEE Std 1364-2001.

`$setuphold`

Combines the `$setup` and `$hold` system tasks. See IEEE Std 1364-1995 page 189 for the official description. There is also an extended syntax that is in IEEE Std 1364-2001 pages 242-244. This extended syntax is as follows:

```
$setuphold(reference_event, data_event,  
           setup_limit, hold_limit, notifier,  
           timestamp_cond, timecheck_cond, delay_reference,  
           delay_data);
```

`$skew`

Reports a timing violation when a reference event happens too long after a data event. See IEEE std 1364-2001 pages 249-250.

`$width`

Reports a timing violation when a pulse is narrower than the specified limit. See IEEE std 1364-2001 pages 254-255. VCS ignores the threshold argument.

System Tasks for PLA Modeling

`$async$and$array to $sync$nor$plane`

See IEEE Std 1364-2001 page 302.

System Tasks for Stochastic Analysis

`$q_add`

Places an entry on a queue in stochastic analysis. See IEEE Std 1364-2001 page 307.

`$q_exam`

Provides statistical information about activity at the queue. See IEEE Std 1364-2001 page 307.

`$q_full`

Returns 0 if the queue is not full, returns a 1 if the queue is full. See IEEE Std 1364-2001 page 307.

`$q_initialize`

Creates a new queue. See IEEE Std 1364-2001 page 306-307.

`$q_remove`

Receives an entry from a queue. See IEEE Std 1364-2001 page 307.

System Tasks for Simulation Time

`$realtime`

Returns a real number time. See IEEE Std 1364-2001 pages 309-310.

`$stime`

Returns an unsigned integer that is a 32-bit time. See IEEE Std 1364-2001 page 309.

`$time`

Returns an integer that is a 64-bit time. See IEEE Std 1364-2001 pages 308-309.

System Tasks for Probabilistic Distribution

`$dist_exponential`

Returns random numbers where the distribution function is exponential. See IEEE std 1364-2001 page 312.

`$dist_normal`

Returns random numbers with a specified mean and standard deviation. See IEEE Std 1364-2001 page 312.

`$dist_poisson`

Returns random numbers with a specified mean. See IEEE Std 1364-2001 page 312.

`$dist_uniform`

Returns random numbers uniformly distributed between parameters. See IEEE Std 1364-2001 page 312.

`$random`

Provides a random number. See IEEE Std 1364-2001 page 312. Using this system function in certain kinds of statements might cause simulation failure.

System Tasks for Resetting VCS

`$reset`

Resets the simulation time to 0. See IEEE Std 1364-2001 pages 741-742.

`$reset_count`

Keeps track of the number of times VCS executes the `$reset` system task in a simulation session. See IEEE std 1364-2001 pages 741-742.

`$reset_value`

System function that you can use to pass a value from, before or after VCS executes the `$reset` system task, that is, you can enter a `reset_value` integer argument to the `$reset` system task, and after VCS resets the simulation, the `$reset_value` system function returns this integer argument. See IEEE std 1364-2001 pages 741-742.

General System Tasks and Functions

Checks for a Plusarg

`$test$plusargs`

Checks for the existence of a given plusarg on the runtime executable command line. Syntax:

`$test$plusargs("plusarg_without_the_+");`

SDF Files

`$sdf_annotation`

Tells VCS to back-annotate delay values from an SDF file to your Verilog design.

Counting the Drivers on a Net

`$countdrivers`

Counts the number of drivers on a net. See IEEE std 1364-2001 page 738-739.

Depositing Values

`$deposit`

Deposits a value on a net or variable. This deposited value overrides the value from any other driver of the net or variable. The value propagates to all loads of the net or variable. A subsequent simulation event can override the deposited value. You cannot use this system task to deposit values to bit-selects or part-selects.

Syntax: `$deposit (net_or_variable, value) ;`

The deposited value can be the value of another net or variable. You can deposit the value of a bit-select or part-select.

Fast Processing Stimulus Patterns

`$getpattern`

Provides for fast processing of stimulus patterns. See IEEE std 1364-2001 page 739.

Saving and Restarting The Simulation State

`$save`

Saves the current simulation state in a file. See IEEE std 1364-2001 pages 742-743.

`$restart`

Restores the simulation to the state that you saved in the check file with the `$save` system task. See IEEE std 1364-2001 pages 742-743.

Checking for X and Z Values in Conditional Expressions

`$xzcheckon`

Displays a warning message every time VCS evaluates a conditional expression to have an X or Z value.

Syntax: `$xzcheckon(level_number, hierarchical_name)`

level_number (Optional)

Specifies the number of hierarchy scope levels from the specified module instance to check for X and Z values. If the number is 0 or not specified, implies to check all scope instances to the end of the hierarchy.

hierarchical_name (Optional)

Hierarchical name of the module instance, that is, the top-level instance of the subhierarchy for which you want to enable checking.

`$xzcheckoff`

Suppress the warning message every time VCS evaluates a conditional expression to have an X or Z value.

Syntax:

`$xzcheckoff(level_number, hierarchical_name)`

level_number (Optional)

Specifies the number of hierarchy scope levels from the specified module instance, for which X and Z value check is disabled. If the number is 0 or not specified, implies to disable the check on all scope instances to the end of the hierarchy.

hierarchical_name (Optional)

Hierarchical name of the module instance, that is, the top-level instance of the subhierarchy for which you want to disable checking.

Calculating Bus Widths

`$clog2`

Use this system function to calculate bus widths from, for example, parameters. The following illustrates its use:

```
integer result;  
result = $clog2(n);
```

Note:

If the argument has x or z values then that bit will be considered as 1 or 0 respectively by VCS. The argument could be a vector with a few bits having x or z values.

For more information on this system function, see section 17.11.1 or the *IEEE Std-1364-2005 Verilog LRM*.

IEEE Standard System Tasks Not Yet Implemented

The following Verilog system tasks are included in the IEEE Std 1364-2001 standards, but are not yet implemented in VCS:

\$dist_chi_square
\$dist_t

\$dist_erlang
\$nochange

E

PLI Access Routines

VCS includes a number of access routines. This appendix describes these access routines in the following sections:

- “Access Routines for Reading and Writing to Memories”
- “Access Routines for Multidimensional Arrays”
- “Access Routines for Probabilistic Distribution”
- “Access Routines for Returning a Pointer to a Parameter Value”
- “Access Routines for Extended VCD Files”
- “Access Routines for Line Callbacks”
- “Access Routines for Source Protection”

- “Access Routine for Signal in a Generate Block”
- “VCS API Routines”

Access Routines for Reading and Writing to Memories

VCS includes a number of access routines for reading and writing to a memory.

These access routines are as follows:

`acc_setmem_int`

Writes an integer value to specific bits in a Verilog memory word.
See “[“acc_setmem_int” on page 5](#) for details.

`acc_getmem_int`

Reads an integer value from specific bits in a Verilog memory word.
See “[“acc_getmem_int” on page 6](#) for details.

`acc_clearmem_int`

Clears a memory, that is, writes zeros to all bits. See
“[“acc_clearmem_int” on page 7](#) for details.

`acc_setmem_hexstr`

Writes a hexadecimal string value to specific bits in a Verilog memory word.
See “[“acc_setmem_hexstr” on page 13](#) for details.

`acc_getmem_hexstr`

Reads a hexadecimal string value from specific bits in a Verilog memory word.
See “[“acc_getmem_hexstr” on page 18](#) for details.

`acc_setmem_bitstr`

Writes a string of binary bits (including x and z) to a Verilog memory word. See “[“acc_setmem_bitstr” on page 19](#) for details.

`acc_getmem_bitstr`

Reads a bit string from specific bits in a Verilog memory word. See “[“acc_getmem_bitstr” on page 20](#) for details.

`acc_handle_mem_by_fullname`

Returns the handle used by `acc_readmem`. See “[“acc_handle_mem_by_fullname” on page 21](#) for details.

`acc_readmem`

Reads a data file and writes the contents to a memory. See “[“acc_readmem” on page 22](#) for details.

`acc_getmem_range`

Returns the upper and lower limits of a memory. See “[“acc_getmem_range” on page 24](#) for details.

`acc_getmem_size`

Returns the number of elements (or words or addresses) in a memory. See “[“acc_getmem_size” on page 25](#) for details.

`acc_getmem_word_int`

Returns the integer of a memory element. See “[“acc_getmem_word_int” on page 26](#) for details.

`acc_getmem_word_range`

Returns the least significant bit of a memory element and the length of the element. See “[“acc_getmem_word_range” on page 27](#) for details.

acc_setmem_int

You use the `acc_setmem_int` access routine to write an integer value to specific bits in a Verilog memory word.

acc_setmem_int		
Synopsis:	Writes an integer value to specific bits in a memory word.	
Syntax:	<code>acc_setmem_int (memhand, value, row, start, length)</code>	
	Type	Description
Returns:	void	
Arguments:	Type	Name Description
	handle	memhand Handle to memory
	int	value The integer value written in binary format to the bits in the word.
	int	row The memory array index.
	int	start Bit number of the leftmost bit in the memory word where this routine starts writing the value.
	int	length Starting with the start bit, specifies the total number of bits this routine writes to.
Related routines:	<code>acc_getmem_int</code> <code>acc_setmem_hexstr</code> <code>acc_getmem_hexstr</code> <code>acc_setmem_bitstr</code> <code>acc_getmem_bitstr</code> <code>acc_clearmem_int</code> <code>acc_handle_mem_by_fullname</code> <code>acc_readmem</code> <code>acc_getmem_range</code> <code>acc_getmem_size</code> <code>acc_getmem_word_int</code> <code>acc_getmem_word_range</code>	

acc_getmem_int

You use the `acc_getmem_int` access routine to return an integer value for certain bits in a Verilog memory word.

acc_getmem_int			
Synopsis:	Returns an integer value for specific bits in a memory word.		
Syntax:	<code>acc_getmem_int (memhand, row, start, length)</code>		
Returns:	Type	Description	
	int	Integer value of the bits in the memory word.	
Arguments:	Type	Name	Description
	handle	memhand	Handle to memory
	int	row	The memory array index
	int	start	Bit number of the leftmost bit in the memory word where this routine starts reading the value.
	int	length	Specifies the total number of bits this routine reads starting with the start bit.
Related routines:	<code>acc_setmem_int</code> <code>acc_setmem_hexstr</code> <code>acc_getmem_hexstr</code> <code>acc_setmem_bitstr</code> <code>acc_getmem_bitstr</code> <code>acc_clearmem_int</code> <code>acc_handle_mem_by_fullname</code> <code>acc_readmem</code> <code>acc_getmem_range</code> <code>acc_getmem_size</code> <code>acc_getmem_word_int</code> <code>acc_getmem_word_range</code>		

acc_clearmem_int

You use the `acc_clearmem_int` access routine to write zeros to all bits in a memory.

acc_clearmem_int			
Synopsis:	Clears a memory word.		
Syntax:	<code>acc_clearmem_int (memhand)</code>		
Returns:	Type	Description	
Arguments:	Type	Name	Description
Related routines:	void		
	handle	memhand	Handle to memory
	<code>acc_setmem_int</code> <code>acc_getmem_int</code> <code>acc_setmem_hexstr</code> <code>acc_getmem_hexstr</code> <code>acc_setmem_bitstr</code> <code>acc_getmem_bitstr</code> <code>acc_handle_mem_by_fullname</code> <code>acc_readmem</code> <code>acc_getmem_range</code> <code>acc_getmem_size</code> <code>acc_getmem_word_int</code> <code>acc_getmem_word_range</code>		

Examples

The following code examples illustrate how to use `acc_getmem_int`, `acc_setmem_int`, and `acc_clearmem_int`:

- [Example E-1](#) shows C code that includes a number of functions to be associated with user-defined system tasks.
- [Example E-2](#) shows the PLI table for associating these functions with these system tasks.

- [Example E-3](#) shows the Verilog source code containing these system tasks.

Example E-1 C Source Code for Functions Calling acc_getmem_int, acc_setmem_int, and acc_clearmem_int

```
#include <stdio.h>
#include "acc_user.h"
#include "vcs_acc_user.h"

void error_handle(char *msg)
{
    printf("%s", msg);
    fflush(stdout);
    exit(1);
}

void set_mem()
{
    handle memhand = NULL;
    int value = -1;
    int row = -1;
    int start_bit = -1;
    int len = -1;

    memhand = acc_handle_tfarg(1);
    if(!memhand) error_handle("NULL MEM HANDLE\n");
    value = acc_fetch_tfarg_int(2);
    row = acc_fetch_tfarg_int(3);
    start_bit = acc_fetch_tfarg_int(4);
    len = acc_fetch_tfarg_int(5);

    acc_setmem_int(memhand, value, row, start_bit, len);
}
```

```

void get_mem()
{
    handle memhand = NULL;
    int row = -1;
    int start_bit = -1;
    int len = -1;
    int value = -1;

    memhand = acc_handle_tfarg(1);
    if(!memhand) error_handle("NULL MEM HANDLE\n");
    row = acc_fetch_tfarg_int(2);
    start_bit = acc_fetch_tfarg_int(3);
    len = acc_fetch_tfarg_int(4);
    value = acc_getmem_int(memhand, row, start_bit, len);
    printf("getmem: value of word %d is : %d\n",row,value);
    fflush(stdout);
}

void clear_mem()
{
    handle memhand = NULL;

    memhand = acc_handle_tfarg(1);
    if(!memhand) error_handle("NULL MEM HANDLE\n");

    acc_clearmem_int(memhand);
}

```

The function with the `set_mem` identifier calls the IEEE standard `acc_fetch_tfarg_int` routine to get the handles for arguments to the user-defined system task that you associate with this function in the PLI table file. It then assigns the handles to local variables and calls `acc_setmem_int` to write to the specified memory in the specified word, start bit, for the specified length.

Similarly, the function with the `get_mem` identifier calls the `acc_fetch_tfarg_int` routine to get the handles for arguments to a user-defined system task and assign them to local variables. It

then calls `acc_gtetmem_int` to read from the specified memory in the specified word, starting with the specified start bit for the specified length. It then displays the word index of the memory and its value.

The function with the `clear_mem` identifier likewise calls the `acc_fetch_tfarg_int` routine to get a handle and then calls `acc_clear_mem_int` with that handle.

Example E-2 PLI Table File

```
$set_mem  call=set_mem acc+=rw:*
$get_mem  call=get_mem acc+=r:*
$clear_mem call=clear_mem acc+=rw:*
```

Here the `$set_mem` user-defined system task is associated with the `set_mem` function in the C code, as are the `$get_mem` and `$clear_mem` with their corresponding `get_mem` and `clear_mem` function identifiers.

Example E-3 Verilog Source Code Using These System Tasks

```
module top;
// read and print out data of memory
parameter start = 0;
parameter finish = 9 ;
parameter bstart =1 ;
parameter bfinish =8 ;
parameter size = finish - start + 1;
reg [bfinish:bstart] mymem[start:finish];
integer i;
integer len;
integer value;

initial
begin
// $set_mem(mem_name, value, row, start_bit, len)
$clear_mem(mymem);

// set values
#1 $set_mem(mymem, 8, 2, 1, 5);
#1 $set_mem(mymem, 32, 3, 1, 6);
#1 $set_mem(mymem, 144, 4, 1, 8);
#1 $set_mem(mymem,29,5,1,8);

// print values through acc_getmem_int
#1 len = bfinish - bstart + 1;
$display();
$display("Begin Memory Values");
for (i=start;i<=finish;i=i+1)
begin
$get_mem(mymem,i,bstart,len);
end
$display("End Memory Values");
$display();

// display values
#1 $display();
$display("Begin Memory Display");
for (i=start;i<=finish;i=i+1)
begin
$display("mymem word %d is %b",i,mymem[i]);
```

```
    end
$display("End Memory Display");
$display();
end
endmodule
```

In this Verilog code, in the initial block, the following events occur:

1. The `$clear_mem` system task clears the memory.
2. Then the `$set_mem` system task deposits values in specified words, and in specified bits in the memory named `mymem`.
3. In a `for` loop, the `$get_mem` system task reads values from the memory and displays those values.

acc_setmem_hexstr

You use the `acc_setmem_hexstr` access routine for writing the corresponding binary representation of a hexadecimal string to a Verilog memory.

acc_setmem_hexstr			
Synopsis:	Writes a hexadecimal string to a word in a Verilog memory.		
Syntax:	<code>acc_setmem_hexstr (memhand, hexStrValue, row, start)</code>		
Returns:	Type	Description	
Arguments:	void		
	Type	Name	Description
	handle	memhand	Handle to memory
	char *	hexStrValue	Hexadecimal string
Related routines:	int	row	The memory array index
	int	start	Bit number of the leftmost bit in the memory word where this routine starts writing the string.
	<code>acc_setmem_int</code> <code>acc_getmem_int</code> <code>acc_getmem_hexstr</code> <code>acc_setmem_bitstr</code> <code>acc_getmem_bitstr</code> <code>acc_clearmem_int</code> <code>acc_handle_mem_by_fullname</code> <code>acc_readmem</code> <code>acc_getmem_range</code> <code>acc_getmem_size</code> <code>acc_getmem_word_int</code> <code>acc_getmem_word_range</code>		

This routine takes a value argument which is a hexadecimal string of any size and puts its corresponding binary representation into the memory word indexed by `row`, starting at the bit number `start`.

Examples

The following code examples illustrates the use of `acc_setmem_hexstr`:

- [Example E-4](#) shows the C source code for an application that calls `acc_setmem_hexstr`.
- [Example E-5](#) shows the contents of a data file read by the application.
- [Example E-6](#) shows the PLI table file that associates the user-defined system task in the Verilog code with the application.
- [Example E-7](#) shows the Verilog source that calls the application.

*Example E-4 C Source Code For an Application Calling
acc_setmem_hexstr*

```
#include <stdio.h>
#include "acc_user.h"
#include "vcsuser.h"
#define NAME_SIZE 256
#define len 100
pli()
{
    FILE *infile;
    char memory_name[NAME_SIZE] ;
    char value[len];
    handle memory_handle;
    int row,start;

    infile = fopen("initfile", "r");
    while ( fscanf(infile,"%s %s %d %d ",
                  memory_name,value,&row,&start) != EOF )
    {
        printf ("The mem= %s \n value= %s \n row= %d \n start= %d \n",
               memory_name,value,row,start);
        memory_handle=acc_handle_object(memory_name);
        acc_setmem_hexstr(memory_handle,value,row,start);
    }
}
```

[Example E-4](#) shows the source code for a PLI application that:

1. Reads a data file named `initfile` to find the memory identifiers of the memories it writes to, the hexadecimal string to be converted to its bit representation when written to the memory, the index of the memory where it writes this value, and the starting bit for writing the binary value.
2. Displays where in the memory it is writing these values
3. Calls the access routine to write the values in the `initfile`.

Example E-5 The Data File Read by the Application

```
testbench.U2.cmd_array 5 0 0
testbench.U2.cmd_array a5 1 4
testbench.U2.cmd_array a5a5 2 8
testbench.U1.slave_addr a073741824 0 4
testbench.U1.slave_addr 16f0612735 1 8
testbench.U1.slave_addr 2b52a90e15 2 12
```

Each line lists a Verilog memory, followed by a hex string, a memory index, and a start bit.

Example E-6 PLI Table File

```
$pli call=pli acc=rw:*
```

Here the \$pli system task is associated with the function with the pli identifier in the C source code.

Example E-7 Verilog Source Calling the PLI Application

```
module testbench;
    monitor U1 ();
    master  U2 ();
    initial begin
        $monitor($stime,,,
            "sladd[0]=%h sladd[1]=%h sladd[2]=%h load=%h
            cmd[0]=%h cmd[1]=%h cmd[2]=%h",
            testbench.U1.slave_addr[0],
            testbench.U1.slave_addr[1],
            testbench.U1.slave_addr[2],
            testbench.U1.load,
            testbench.U2.cmd_array[0],
            testbench.U2.cmd_array[1],
            testbench.U2.cmd_array[2] );
        #10;
        $pli();
    end
endmodule
```

```

module master;
    reg[31:0] cmd_array [0:2];
    integer i;
initial begin      //setup some default values
    for (i=0; i<3; i=i+1)
        cmd_array[i] = 32'h0000_0000;
end
endmodule

module monitor;
    reg load;
    reg[63:0] slave_addr [0:2];
    integer i;
initial begin //setup some default values
    for (i=0; i<3; i=i+1)
        slave_addr[i] = 64'h0000_0000_0000_0000;
    load = 1'b0;
end
endmodule

```

In Example E-7 module testbench calls the application using the \$pli user-defined system task for the application. The display string in the \$monitor system task is on two lines to enhance readability.

acc_getmem_hexstr

You use the `acc_getmem_hexstr` access routine to get a hexadecimal string from a Verilog memory.

acc_getmem_hexstr			
Synopsis:	Returns a hexadecimal string from a Verilog memory.		
Syntax:	<code>acc_getmem_hexstr (memhand, hexStrValue, row, start, len)</code>		
	Type	Description	
Returns:	void		
Arguments:	Type	Name	Description
	handle	memhand	Handle to memory
	char *	hexStrValue	Pointer to a character array into which the string is written
	int	row	The memory array index
	int	start	Bit number of the leftmost bit in the memory word where this routine starts reading the string.
	int	length	Specifies the total number of bits this routine reads starting with the start bit.
Related routines:	<code>acc_setmem_int</code> <code>acc_getmem_int</code> <code>acc_setmem_hexstr</code> <code>acc_setmem_bitstr</code> <code>acc_getmem_bitstr</code> <code>acc_clearmem_int</code> <code>acc_handle_mem_by_fullname</code> <code>acc_readmem</code> <code>acc_getmem_range</code> <code>acc_getmem_size</code> <code>acc_getmem_word_int</code> <code>acc_getmem_word_range</code>		

acc_setmem_bitstr

You use the `acc_setmem_bitstr` access routine for writing a string of binary bits (including x and z) to a Verilog memory.

acc_setmem_bitstr			
Synopsis:	Writes a string of binary bits to a word in a Verilog memory.		
Syntax:	<code>acc_setmem_bitstr (memhand, bitStrValue, row, start)</code>		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	handle	memhand	Handle to memory
	char *	bitStrValue	Bit string
	int	row	The memory array index
	int	start	Bit number of the leftmost bit in the memory word where this routine starts writing the string.
Related routines:	<code>acc_setmem_int</code> <code>acc_getmem_int</code> <code>acc_setmem_hexstr</code> <code>acc_getmem_hexstr</code> <code>acc_getmem_bitstr</code> <code>acc_clearmem_int</code> <code>acc_handle_mem_by_fullname</code> <code>acc_readmem</code> <code>acc_getmem_range</code> <code>acc_getmem_size</code> <code>acc_getmem_word_int</code> <code>acc_getmem_word_range</code>		

This routine takes a value argument that is a bit string of any size, which can include the x and z values, and puts its corresponding binary representation into the memory word indexed by `row`, starting at the bit number `start`.

acc_getmem_bitstr

You use the `acc_getmem_bitstr` access routine to get a bit string, including x and z values, from a Verilog memory.

acc_getmem_bitstr			
Synopsis:	Returns a hexadecimal string from a Verilog memory.		
Syntax:	<code>acc_getmem_bitstr (memhand,bitStrValue,row,start,len)</code>		
	Type	Description	
Returns:	void		
Arguments:	Type	Name	Description
	handle	memhand	Handle to memory
	char *	hexStrValue	Pointer to a character array into which the string is written
	int	row	The memory array index
	int	start	Bit number of the leftmost bit in the memory word where this routine starts reading the string.
	int	length	Specifies the total number of bits this routine reads starting with the start bit.
Related routines:	<code>acc_setmem_int</code> <code>acc_getmem_int</code> <code>acc_setmem_hexstr</code> <code>acc_getmem_hexstr</code> <code>acc_setmem_bitstr</code> <code>acc_clearmem_int</code> <code>acc_handle_mem_by_fullname</code> <code>acc_readmem</code> <code>acc_getmem_range</code> <code>acc_getmem_size</code> <code>acc_getmem_word_int</code> <code>acc_getmem_word_range</code>		

acc_handle_mem_by_fullname

Returns a handle to a memory that can only be used as a parameter to acc_readmem.

acc_handle_mem_by_fullname		
Synopsis:	Returns a handle to be used as a parameter to acc_readmem only	
Syntax:	acc_handle_mem_by_fullname (fullMemInstName)	
	Type	Description
Returns:	handle	Handle to the instance
	Type	Name
Arguments:	char*	fullMemInstName
		Hierarchical name for a memory
Related routines:	acc_setmem_int acc_getmem_int acc_setmem_hexstr acc_getmem_hexstr acc_setmem_bitstr acc_getmem_bitstr acc_handle_mem_by_fullname acc_readmem acc_getmem_range acc_getmem_size acc_getmem_word_int acc_getmem_word_range	

acc_readmem

You use the `acc_readmem` access routine to read a data file into a memory. It is similar to the `$readmemb` or `$readmemh` system tasks.

The `memhandle` argument must be the handle returned by `acc_handle_mem_by_fullname`.

acc_readmem			
Synopsis:	Reads a data file into a memory		
Syntax:	<code>acc_readmem (memhandle, data_file, format)</code>		
Returns:	Type	Description	
Arguments:	void		
	Type	Name	Description
Arguments:	handle	memhandle	Handle returned by <code>acc_handle_mem_fullname</code>
	const char*	data_file	Data file this routine reads
	int	format	Specify a character that is promoted to int. 'h' for hexadecimal data, 'b' for binary data.
Related routines:	<code>acc_setmem_int</code> <code>acc_getmem_int</code> <code>acc_setmem_hexstr</code> <code>acc_getmem_hexstr</code> <code>acc_setmem_bitstr</code> <code>acc_getmem_bitstr</code> <code>acc_handle_mem_by_fullname</code> <code>acc_readmem</code> <code>acc_getmem_range</code> <code>acc_getmem_size</code> <code>acc_getmem_word_int</code> <code>acc_getmem_word_range</code>		

Examples

The following code examples illustrate the use of `acc_readmem` and `acc_handle_mem_by_fullname`.

Example E-8 C Source Code Calling Tacc_readmem and acc_handle_mem_by_fullname

```
#include "acc_user.h"
#include "vcs_acc_user.h"
#include "vcsuser.h"

int test_acc_readmem(void)
{
    const char *memName = tf_getcstringp(1);
    const char *memFile = tf_getcstringp(2);
    handle mem = acc_handle_mem_by_fullname(memName);

    if (mem) {
        io_printf("test_acc_readmem: %s handle found\n",
memName);
        acc_readmem(mem, memFile, 'h');
    }
    else {
        io_printf("test_acc_readmem: %s handle NOT found\n",
memName);
    }
}
```

Example E-9 The PLI Table File

```
$test_acc_readmem call=test_acc_readmem
```

Example E-10 The Verilog Source Code

```
module top;
reg [7:0] CORE[7:0];
initial $acc_readmem(CORE, "CORE");
initial $test_acc_readmem("top.CORE", "test_mem_file");
endmodule
```

acc_getmem_range

You use the acc_getmem_range access routine to access the upper and lower limits of a memory.

acc_getmem_range						
Synopsis:	Returns the upper and lower limits of a memory					
Syntax:	acc_getmem_range (memhandle, p_left_index,p_right_index)					
Returns:	<table><thead><tr><th>Type</th><th>Description</th></tr></thead><tbody><tr><td>void</td><td></td></tr></tbody></table>		Type	Description	void	
Type	Description					
void						
Arguments:	Type	Name Description				
	handle	memhandle Handle to a memory				
	int*	p_left_index Pointer to int				
	int	p_right_index Pointer to int				
Related routines:	acc_setmem_int acc_getmem_int acc_setmem_hexstr acc_getmem_hexstr acc_setmem_bitstr acc_getmem_bitstr acc_handle_mem_by_fullname acc_readmem acc_getmem_size acc_getmem_word_int acc_getmem_word_range					

acc_getmem_size

You use the `acc_getmem_size` access routine to access the number of elements in a memory.

acc_getmem_size		
Synopsis:	Returns the number of elements in a memory	
Syntax:	<code>acc_getmem_size (memhandle)</code>	
	Type	Description
Returns:	int	The number of elements in a memory
	Type	Name
Arguments:	handle	memhandle
Related routines:	Handle to a memory	
	acc_setmem_int	
	acc_getmem_int	
	acc_setmem_hexstr	
	acc_getmem_hexstr	
	acc_setmem_bitstr	
	acc_getmem_bitstr	
	acc_handle_mem_by_fullname	
	acc_readmem	
	acc_getmem_range	

acc_getmem_word_int

You use the `acc_getmem_word_int` access routine to access the integer value of an element (or word, address, or row).

acc_getmem_word_int		
Synopsis:	Returns the integer value of an element	
Syntax:	<code>acc_getmem_word_int (memhandle, row)</code>	
	Type	Description
Returns:	int	The integer value of a row
	Type	Name
Arguments:	handle	memhandle
	int	row
Handle to a memory The element (word address, or row) in the memory		
Related routines:	<code>acc_setmem_int</code> <code>acc_getmem_int</code> <code>acc_setmem_hexstr</code> <code>acc_getmem_hexstr</code> <code>acc_setmem_bitstr</code> <code>acc_getmem_bitstr</code> <code>acc_handle_mem_by_fullname</code> <code>acc_readmem</code> <code>acc_getmem_range</code> <code>acc_getmem_size</code> <code>acc_getmem_word_range</code>	

acc_getmem_word_range

You use the acc_getmem_word_range access routine to access the least significant bit of an element (or word, address, or row) and the length of the element.

acc_getmem_word_range						
Synopsis:	Returns the least significant bit of an element and the length of the element					
Syntax:	acc_getmem_word_range (memhandle, lsb, len)					
Returns:	<table><thead><tr><th>Type</th><th>Description</th></tr></thead><tbody><tr><td>void</td><td></td></tr></tbody></table>		Type	Description	void	
Type	Description					
void						
Arguments:	Type	Name Description				
	handle	memhandle Handle to a memory				
	int*	lsb Pointer to the least significant bit				
	int*	len Pointer to the length of the element				
Related routines:	acc_setmem_int acc_getmem_int acc_setmem_hexstr acc_getmem_hexstr acc_setmem_bitstr acc_getmem_bitstr acc_handle_mem_by_fullname acc_readmem acc_getmem_range acc_getmem_size acc_getmem_word_int					

Access Routines for Multidimensional Arrays

The type for multi-dimensional arrays is defined in the `vcs_acc_user.h` file. Its name is `accMda`.

We also have the following tf and access routines for accessing data in a multi-dimensional array:

`tf_mdanodeinfo` and `tf_imdanodeinfo`

Returns access parameter node information from a multi-dimensional array. See “[tf_mdanodeinfo and tf_imdanodeinfo](#)” on page 29 for details.

`acc_get_mda_range`

Returns all the ranges of the multi-dimensional array. See “[acc_get_mda_range](#)” on page 31 for details.

`acc_get_mda_word_range`

Returns the range of an element in a multi-dimensional array. See “[acc_get_mda_word_range\(\)](#)” on page 32 for details.

`acc_getmda_bitstr`

Reads a bit string, including X and Z values, from an element in a multi-dimensional array. See “[acc_getmda_bitstr\(\)](#)” on page 34 for details.

`acc_setmda_bitstr`

Writes a bit string, including X and Z values, from an element in a multi-dimensional array. See “[acc_setmda_bitstr\(\)](#)” on page 35 for details.

tf_mdanodeinfo and tf_imdanodeinfo

You use these routines to access parameter node information from a multi-dimensional array.

tf_mdanodeinfo(), tf_imdanodeinfo()			
Synopsis:	Returns access parameter node information from a multi-dimensional array.		
Syntax:	<code>tf_mdanodeinfo(nparam, mdanodeinfo_p)</code> <code>tf_imdanodeinfo(nparam, mdanodeinfo_p, instance_p)</code>		
	Type	Description	
Returns:	<code>mdanodeinfo_p *</code>	The value of the second argument if successful; 0 if an error occurs	
	Type	Name	Description
Arguments:	<code>int</code>	<code>nparam</code>	Index number of the multi-dimensional array parameter
	<code>struct t_tfmdanodeinfo *</code>	<code>mdanodeinfo_p</code>	Pointer to a variable declared as the <code>t_tfmdanodeinfo</code> structure type
	<code>char *</code>	<code>instance_p</code>	Pointer to a specific instance of a multi-dimensional array
Related routines:	<code>acc_get_mda_range</code> <code>acc_get_mda_word_range</code> <code>acc_getmda_bitstr</code> <code>acc_setmda_bitstr</code>		

Structure t_tfmdanodeinfo is defined in the vcsuser.h file as follows:

```
typedef struct t_tfmdanodeinfo
{
    short node_type;
    short node_fulltype;
    char *memoryval_p;
    char *node_symbol;
    int node_ngroups;
    int node_vec_size;
    int node_sign;
    int node_ms_index;
    int node_ls_index;
    int node_mem_size;
    int *node_lhs_element;
    int *node_rhs_element;
    int node_dimension;
    int *node_handle;
    int node_vec_type;
} s_tfmdanodeinfo, *p_tfmdanodeinfo;
```

acc_get_mda_range

The `acc_get_mda_range` routine returns the ranges of a multi-dimensional array.

acc_get_mda_range()			
Synopsis:	Gets all the ranges of the multi-dimensional array.		
Syntax:	<code>acc_get_mda_range(mdaHandle, size, msb, lsb, dim, plndx, prndx)</code>		
	Type	Description	
Returns:	<code>void</code>		
Arguments:	Type	Name	Description
	<code>handle</code>	<code>mdaHandle</code>	Handle to the multi-dimensional array
	<code>int *</code>	<code>size</code>	Pointer to the size of the multi-dimensional array
	<code>int *</code>	<code>msb</code>	Pointer to the most significant bit of a range
	<code>int *</code>	<code>lsb</code>	Pointer to the least significant bit of a range
	<code>int *</code>	<code>dim</code>	Pointer to the number of dimensions in the multi-dimensional array
	<code>int *</code>	<code>plndx</code>	Pointer to the left index of a range
	<code>int *</code>	<code>prndx</code>	Pointer to the right index of a range
Related routines:	<code>tf_mdanodeinfo</code> and <code>tf_imdanodeinfo</code> <code>acc_get_mda_word_range</code> <code>acc_getmda_bitstr</code> <code>acc_setmda_bitstr</code>		

If you have a multi-dimensional array such as the following:

```
reg [7:0] my_mem[255:0][255:0][31:0];
```

And you call a routine, such as the following:

```
handle hN = acc_handle_by_name(my_mem);  
acc_get_mda_range(hN, &size, &msb, &lsb, &dim, &plndx,  
&prndx);
```

It yields the following result:

```
size = 8;  
msb = 7, lsb = 0;  
dim = 4;  
plndx[] = {255, 255, 31}  
prndx[] = {0, 0, 0}
```

acc_get_mda_word_range()

The `acc_get_mda_word_range` routine returns the range of an element in a multi-dimensional array.

acc_get_mda_word_range()			
Synopsis:	Gets the range of an element in a multi-dimensional array.		
Syntax:	<code>acc_get_mda_range(mdaHandle, msb, lsb)</code>		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	handle	mdaHandle	Handle to the multi-dimensional array
	int *	msb	Pointer to the most significant bit of a range
	int *	lsb	Pointer to the least significant bit of a range
Related routines:	<code>tf_mdanodeinfo</code> and <code>tf_imdanodeinfo</code> <code>acc_get_mda_range</code> <code>acc_getmda_bitstr</code> <code>acc_setmda_bitstr</code>		

If you have a multi-dimensional array such as the following:

```
reg [7:0] my_mem[255:0][255:0][31:0];
```

And you call a routine, such as the following:

```
handle hN = acc_handle_by_name(my_mem);  
acc_get_mda_word_range(hN, &left, &right);
```

It yields the following result:

```
left = 7;  
right = 0;
```

acc_getmda_bitstr()

You use the `acc_getmda_bitstr` access routine to read a bit string, including x and z values, from a multi-dimensional array.

acc_getmda_bitstr()		
Synopsis:	Gets a bit string from a multi-dimensional array.	
Syntax:	<code>acc_getmda_bitstr(mdaHandle, bitStr, dim, start, len)</code>	
Returns:	Type Description	
Arguments:	Type	Name Description
	handle	mdaHandle Handle to the multi-dimensional array
	char *	bitStr Pointer to the bit string
	int *	dim Pointer to the dimension in the multi-dimensional array
	int *	start Pointer to the start element in the dimension
	int *	len Pointer to the length of the string
Related routines:	<code>tf_mdanodeinfo</code> and <code>tf_imdanodeinfo</code> <code>acc_get_mda_range</code> <code>acc_get_mda_word_range</code> <code>acc_setmda_bitstr</code>	

If you have a multi-dimensional array such as the following:

```
reg [7:0] my_mem[255:0][255:0][31:0];
```

And you call a routine, such as the following:

```
dim[]={5, 5, 10};  
handle hN = acc_handle_by_name(my_mem);  
acc_getmda_bitstr(hN, &bitStr, dim, 3, 3);
```

It yields the following string from `my_mem[5][5][10][3:5]`.

acc_setmda_bitstr()

You use the `acc_setmda_bitstr` access routine to write a bit string, including x and z values, into a multi-dimensional array.

acc_setmda_bitstr()			
Synopsis:	Sets a bit string in a multi-dimensional array.		
Syntax:	<code>acc_setmda_bitstr(mdaHandle, bitStr, dim, start, len)</code>		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	handle	mdaHandle	Handle to the multi-dimensional array
	char *	bitStr	Pointer to the bit string
	int *	dim	Pointer to the dimension in the multi-dimensional array
	int *	start	Pointer to the start element in the dimension
	int *	len	Pointer to the length of the string
Related routines:	<code>tf_mdanodeinfo</code> and <code>tf_imdanodeinfo</code> <code>acc_get_mda_range</code> <code>acc_get_mda_word_range</code> <code>acc_getmda_bitstr</code>		

If you have a multi-dimensional array such as the following:

```
reg [7:0] my_mem[255:0][255:0][31:0];
```

And you call a routine, such as the following:

```
dim[]={5, 5, 10};  
bitstr="111";  
handle hN = acc_handle_by_name(my_mem);  
acc_setmda_bitstr(hN, &bitStr, dim, 3, 3);
```

It writes 111 in my_mem[5] [5] [10] [3:5].

Access Routines for Probabilistic Distribution

VCS includes the following API routines that duplicate the behavior of the Verilog system functions for probabilistic distribution:

vcs_random

Returns a random number and takes no argument. See “[vcs_random](#)” on page 37 for details.

vcs_random_const_seed

Returns a random number and takes an integer argument. See “[vcs_random_const_seed](#)” on page 38 for details.

vcs_random_seed

Returns a random number and takes a pointer to integer argument. See “[vcs_random_seed](#)” on page 38 for details.

vcs_dist_uniform

Returns random numbers uniformly distributed between parameters. See “[vcs_dist_uniform](#)” on page 39 for details.

`vcs_dist_normal`

Returns random numbers with a specified mean and standard deviation. See “[vcs_dist_normal](#)” on page 40 for details.

`vcs_dist_exponential`

Returns random numbers where the distribution function is exponential. See “[vcs_dist_exponential](#)” on page 41 for details.

`vcs_dist_poisson`

Returns random numbers with a specified mean. See “[vcs_random](#)” on page 37 for details.

These routines are declared in the `vcs_acc_user.h` file in the `$VCS_HOME/lib` directory.

vcs_random

You use this routine to obtain a random number.

vcs_random()		
Synopsis:	Returns a random number.	
Syntax:	<code>vcs_random()</code>	
>Returns:	Type	Description
	<code>int</code>	Random number
Arguments:	Type	Name Description
	<code>None</code>	
Related routines:	<code>vcs_random_const_seed</code> <code>vcs_random_seed</code> <code>vcs_dist_uniform</code> <code>vcs_dist_normal</code> <code>vcs_dist_exponential</code> <code>vcs_dist_poisson</code>	

vcs_random_const_seed

You use this routine to return a random number and you supply an integer constant argument as the seed for the random number.

vcs_random_const_seed			
Synopsis:	Returns a random number.		
Syntax:	vcs_random_const_seed(integer)		
	Type	Description	
Returns:	int	Random number	
	Type	Name	Description
Arguments:	int	integer	An integer constant.
Related routines:	vcs_random vcs_random_seed vcs_dist_uniform vcs_dist_normal vcs_dist_exponential vcs_dist_poisson		

vcs_random_seed

You use this routine to return a random number and you supply a pointer argument.

vcs_random_seed()			
Synopsis:	Returns a random number.		
Syntax:	vcs_random_seed(seed)		
	Type	Description	
Returns:	int	Random number	
	Type	Name	Description
Arguments:	int *	seed	Pointer to an int type.
Related routines:	vcs_random vcs_random_const_seed vcs_dist_uniform vcs_dist_normal vcs_dist_exponential vcs_dist_poisson		

vcs_dist_uniform

You use this routine to return a random number uniformly distributed between parameters.

vcs_dist_uniform			
Synopsis:	Returns random numbers uniformly distributed between parameters.		
Syntax:	<code>vcs_dist_uniform(seed, start, end)</code>		
Returns:	Type	Description	
Arguments:	int	Random number	
	Type	Name	Description
	int *	seed	Pointer to a seed integer value.
	int	start	Starting parameter for distribution range.
	int	end	Ending parameter for distribution range.
Related routines:	vcs_random vcs_random_const_seed vcs_random_seed vcs_dist_normal vcs_dist_exponential vcs_dist_poisson		

vcs_dist_normal

You use this routine to return a random number with a specified mean and standard deviation.

vcs_dist_normal			
Synopsis:	Returns random numbers with a specified mean and standard deviation.		
Syntax:	vcs_dist_normal(seed, mean, standard_deviation)		
Returns:	Type	Description	
	int	Random number	
Arguments:	Type	Name	Description
	int *	seed	Pointer to a seed integer value.
	int	mean	An integer that is the average value of the possible returned random numbers.
	int	standard_deviation	An integer that is the standard deviation from the mean for the normal distribution.
Related routines:	vcs_random vcs_random_const_seed vcs_random_seed vcs_dist_uniform vcs_dist_exponential vcs_dist_poisson		

vcs_dist_exponential

You use this routine to return a random number where the distribution function is exponential.

vcs_dist_exponential			
Synopsis:	Returns random numbers where the distribution function is exponential.		
Syntax:	<code>vcs_dist_exponential(seed, mean)</code>		
	Type	Description	
Returns:	int	Random number	
	Type	Name	Description
Arguments:	int *	seed	Pointer to a seed integer value.
	int	mean	An integer that is the average value of the possible returned random numbers.
Related routines:	vcs_random vcs_random_const_seed vcs_random_seed vcs_dist_uniform vcs_dist_normal vcs_dist_poisson		

vcs_dist_poisson

You use this routine to return a random number with a specified mean.

vcs_dist_poisson			
Synopsis:	Returns random numbers with a specified mean.		
Syntax:	vcs_dist_poisson(seed, mean)		
Returns:	Type	Description	
Arguments:	int	Random number	
	Type	Name	Description
	int *	seed	Pointer to a seed integer value.
	int	mean	An integer that is the average value of the possible returned random numbers.
Related routines:	vcs_random vcs_random_const_seed vcs_random_seed vcs_dist_uniform vcs_dist_normal vcs_dist_exponential		

Access Routines for Returning a Pointer to a Parameter Value

The 1364 Verilog standard states that for access routine `acc_fetch_paramval`, you can cast the return value to a character pointer using the C language cast operators `(char*) (int)`. For example:

```
str_ptr=(char*)(int)acc_fetch_paramval(...);
```

In 64-bit simulation, you should use `long` instead of `int`:

```
str_ptr=(char*)(long)acc_fetch_paramval(...);
```

For your convenience, VCS provides the `acc_fetch_paramval_str` routine to directly return a string pointer.

acc_fetch_paramval_str

Returns the value of a string parameter directly as `char*`.

acc_fetch_paramval_str			
Synopsis:	Returns the value of a string parameter directly as <code>char*</code> .		
Syntax:	<code>acc_fetch_paramval_str(param_handle)</code>		
	Type	Description	
Returns:	<code>char*</code>	string pointer	
	Type	Name	Description
Arguments:	<code>handle</code>	<code>param_handle</code>	Handle to a module parameter or specparam.
Related routines:	<code>acc_fetch_paramval</code>		

Access Routines for Extended VCD Files

VCS provides the following routines to monitor the port activity of a device:

`acc_lsi_dumpports_all`

Adds a checkpoint to the file. See “[acc_lsi_dumpports_all](#)” on [page 45](#) for details.

`acc_lsi_dumpports_call`

Monitors instance ports. See “[acc_lsi_dumpports_call](#)” on [page 46](#) for details.

`acc_lsi_dumpports_close`

Closes specified VCDE files. See “[“acc_lsi_dumpports_close” on page 48](#) for details.

`acc_lsi_dumpports_flush`

Flushes cached data to the VCDE file on disk. See “[“acc_lsi_dumpports_flush” on page 49](#) for details.

`acc_lsi_dumpports_limit`

Sets the maximum VCDE file size. See “[“acc_lsi_dumpports_limit” on page 50](#) for details.

`acc_lsi_dumpports_misc`

Processes miscellaneous events. See “[“acc_lsi_dumpports_misc” on page 52](#) for details.

`acc_lsi_dumpports_off`

Suspends VCDE file dumping. See “[“acc_lsi_dumpports_off” on page 53](#) for details.

`acc_lsi_dumpports_on`

Resumes VCDE file dumping. See “[“acc_lsi_dumpports_on” on page 55](#) for details.

`acc_lsi_dumpports_setformat`

Specifies the format of the VCDE file. See “[“acc_lsi_dumpports_setformat” on page 56](#) for details.

```
acc_lsi_dumpports_vhdl_enable
```

Enables or disables the inclusion of VHDL drivers in the determination of driver values. See “[acc_lsi_dumpports_vhdl_enable](#)” on page 58 for details.

acc_lsi_dumpports_all

Syntax

```
int acc_lsi_dumpports_all(char *filename)
```

Synopsis

Adds a checkpoint to the file.

This is a PLI interface to the \$dumpportsall system task. If the filename argument is NULL, this routine adds a checkpoint to all open VCDE files.

Returns

The number of VCDE files that matched.

Example E-11 Example of acc_lsi_dumpports_all

```
#include "acc_user.h"
#include "vcs_acc_user.h"

handle instance = acc_handle_by_name("test_bench.device",
```

```

0) ;
char *outfile = "device.evcd";

/* use IEEE format for this file */
acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_IEEE);
if (acc_lsi_dumpports_call(instance, outfile)) {
/* rut-roh, error ... */
}
acc_lsi_dumpports_limit(100000, outfile);
...

if (time == yada_yada)
    acc_lsi_dumpports_off(outfile);
    ...

if (time == yada_yada_yada) {
    /* add checkpoint (no need to enable dumping) */
    acc_lsi_dumpports_all(outfile);
    acc_lsi_dumpports_flush(outfile);
}
    ...

if (resume_dumping_now)
    acc_lsi_dumpports_on(outfile);
    ...

```

Caution

This routine may affect files opened by the \$dumpports and \$lsi_dumpports system tasks.

acc_lsi_dumpports_call

Syntax

```
int acc_lsi_dumpports_call(handle instance, char *filename)
```

Synopsis

Monitors instance ports.

This is a PLI interface to the \$lsi_dumpports task. The default file format is the original LSI format, but you can select the IEEE format by calling the routine acc_lsi_dumpports_setformat() prior to calling this routine. Your tab file will need the following acc permissions:

```
acc=cbka, cbk, cbkv: [<instance_name>|*] .
```

Returns

Zero on success, non-zero otherwise. VCS displays error messages through tf_error(). A common error is specifying a file name also being used by a \$dumpports or \$lsi_dumpports system task.

Example E-12 Example of acc_lsi_dumpports_all

```
#include "acc_user.h"
#include "vcs_acc_user.h"

handle instance = acc_handle_by_name("test_bench.device",
0);
char *outfile = "device.evcd";

acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_IEEE);

if (acc_lsi_dumpports_call(instance, outfile)) {
    /* error */
}
```

Caution

Multiple calls to this routine are allowed, but the output file name must be unique for each call.

For proper dumpports operation, your task's miscellaneous function must call acc_lsi_dumpports_misc() with every call it gets. This ensures that the dumpports routines sees all of the simulation

events needed for proper update and closure of the dumports (extended VCD) files. For example, your miscellaneous routine would do the following:

```
my_task_misc(int data, int reason)
{
    acc_lsi_dumports_misc(data, reason);
    ...
}
```

acc_lsi_dumports_close

Syntax

```
int acc_lsi_dumports_call(handle instance, char *filename)
```

Synopsis

Closes specified VCDE files.

This routine reads the list of files opened by a call to the system tasks \$dumports and \$lsi_dumports or the routine `acc_lsi_dumports_call()` and closes all that match either the specified instance handle or the `filename` argument.

One or both arguments can be used. If the instance handle is non-null, this routine closes all files opened for that instance.

Returns

The number of files closed.

Example E-13 Example of acc_lsi_dumports_close

```
#include "acc_user.h"
#include "vcs_acc_user.h"

handle instance = acc_handle_by_name("test_bench.device",
```

```

0) ;
char *outfile1 = "device.evcd1";
char *outfile2 = "device.evcd2";

acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_LSI);

acc_lsi_dumpports_call(instance, outfile1);
acc_lsi_dumpports_call(instance, outfile2);
...
acc_lsi_dumpports_close(NULL, outfile1);
...
acc_lsi_dumpports_close(NULL, outfile2);

```

Caution

A call to this function can also close files opened by the \$lsi_dumpports or \$dumpports system tasks.

acc_lsi_dumpports_flush

Syntax

```
int acc_lsi_dumpports_flush(char *filename)
```

Synopsis

Flushes cached data to the VCDE file on disk.

This is a PLI interface to the \$dumpportsflush system task. If the filename is NULL all open files are flushed.

Returns

The number of files matched.

Example E-14 Example of acc_lsi_dumpports_flush

```
#include "acc_user.h"
#include "vcs_acc_user.h"

handle instance = acc_handle_by_name("test_bench.device",
0);
char *outfile = "device.evcd";

/* use IEEE format for this file */
acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_IEEE);
if (acc_lsi_dumpports_call(instance, outfile)) {
    /* rut-roh */
}
acc_lsi_dumpports_limit(100000, outfile);
...

if (time == yada_yada)
    acc_lsi_dumpports_off(outfile);
...

if (time == yada_yada_yada) {
    /* add checkpoint (no need to enable dumping) */
    acc_lsi_dumpports_all(outfile);
    acc_lsi_dumpports_flush(outfile);
}
...

if (resume_dumping_now)
    acc_lsi_dumpports_on(outfile);
...
```

acc_lsi_dumpports_limit

Syntax

```
int acc_lsi_dumpports_limit(unsigned long filesize, char
*filename)
```

Synopsis

Sets the maximum VCDE file size.

This is a PLI interface to the \$dumpportslimit task. If the filename is NULL, the file size is applied to all files.

Returns

The number of files matched.

Example E-15 Example of acc_lsi_dumpports_limit

```
#include "acc_user.h"
#include "vcs_acc_user.h"

handle instance = acc_handle_by_name("test_bench.device",
```

```

0) ;
char *outfile = "device.evcd";

/* use IEEE format for this file */
acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_IEEE);
if (acc_lsi_dumpports_call(instance, outfile)) {
    /* rut-roh */
}
acc_lsi_dumpports_limit(100000, outfile);
...

if (time == yada_yada)
    acc_lsi_dumpports_off(outfile);
...

if (time == yada_yada_yada) {
    /* add checkpoint (no need to enable dumping) */
    acc_lsi_dumpports_all(outfile);
    acc_lsi_dumpports_flush(outfile);
}
...

if (resume_dumping_now)
    acc_lsi_dumpports_on(outfile);
...

```

Caution

This routine may affect files opened by the \$dumpports and \$lsi_dumpports system tasks.

acc_lsi_dumpports_misc

Syntax

```
void acc_lsi_dumpports_misc(int data, int reason)
```

Synopsis

Processes miscellaneous events.

This is a companion routine for `acc_lsi_dumpports_call()`.

For proper dumpports operation, your task's miscellaneous function must call this routine for each call it gets.

Returns

No return value.

Example E-16 Example or acc_lsi_dumpports_misc

```
#include "acc_user.h"
#include "vcs_acc_user.h"

void my_task_misc(int data, int reason)
{
    acc_lsi_dumpports_misc(data, reason);
    ...
}
```

acc_lsi_dumpports_off

Syntax

```
int acc_lsi_dumpports_off(char *filename)
```

Synopsis

Suspends VCDE file dumping.

This is a PLI interface to the `$dumpportsoff` system task. If the file name is NULL, dumping is suspended on all open files.

Returns

The number of files that matched.

Example E-17 of acc_lsi_dumpports_off Example

```
#include "acc_user.h"
#include "vcs_acc_user.h"

handle instance = acc_handle_by_name("test_bench.device",
0);
char *outfile = "device.evcd";

/* use IEEE format for this file */
acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_IEEE);
if (acc_lsi_dumpports_call(instance, outfile)) {
    /* rut-roh */
}
acc_lsi_dumpports_limit(100000, outfile);
...

if (time == yada_yada)
    acc_lsi_dumpports_off(outfile);
...

if (time == yada_yada_yada) {
    /* add checkpoint (no need to enable dumping) */
    acc_lsi_dumpports_all(outfile);
    acc_lsi_dumpports_flush(outfile);
}
...

if (resume_dumping_now)
    acc_lsi_dumpports_on(outfile);
...
```

Caution

This routine may suspend dumping on files opened by the \$dumpports and \$lsi_dumpports system tasks.

acc_lsi_dumpports_on

Syntax

```
int acc_lsi_dumpports_on(char *filename)
```

Synopsis

Resumes VCDE file dumping.

This is a PLI interface to the \$dumpportson system task. If the filename is NULL, dumping is resumed on all open files.

Returns

The number of files that matched.

Example E-18 Example of acc_lsi_dumpports_on

```
#include "acc_user.h"
#include "vcs_acc_user.h"

handle instance = acc_handle_by_name("test_bench.device",
```

```

0) ;
char *outfile = "device.evcd";

/* use IEEE format for this file */
acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_IEEE);
if (acc_lsi_dumpports_call(instance, outfile)) {
    /* rut-roh */
}
acc_lsi_dumpports_limit(100000, outfile);
...

if (time == yada_yada)
    acc_lsi_dumpports_off(outfile);
...

if (time == yada_yada_yada) {
    /* add checkpoint (no need to enable dumping) */
    acc_lsi_dumpports_all(outfile);
    acc_lsi_dumpports_flush(outfile);
}
...

if (resume_dumping_now)
    acc_lsi_dumpports_on(outfile);
...

```

Caution

This routine may resume dumping on files opened by the \$dumpports and \$lsi_dumpports system tasks.

acc_lsi_dumpports_setformat

Syntax

```
int acc_lsi_dumpports_setformat(lsi_dumpports_format_type
format)
```

Where the valid `lsi_dumpports_format_types` are as follows:

`USE_DUMPPORTS_FORMAT_IEEE`

`USE_DUMPPORTS_FORMAT_LSI`

Synopsis

Specifies the format of the VCDE file.

Use this routine to specify which output format (IEEE or the original LSI) should be used. This routine must be called before `acc_lsi_dumpports_call()`.

Returns

Zero if success, non-zero if error. Errors are reported through `tf_error()`.

Example E-19 Example of acc_lsi_dumpports_setformat

```
#include "acc_user.h"
#include "vcs_acc_user.h"

handle instance = acc_handle_by_name("test_bench.device",
0);
char *outfile1 = "device.evcd1";
char *outfile2 = "device.evcd2";

/* use IEEE format for this file */
acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_IEEE);
if (acc_lsi_dumpports_call(instance, outfile1)) {
    /* error */
}

/* use LSI format for this file */
acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_LSI);
if (acc_lsi_dumpports_call(instance, outfile2)) {
    /* error */
}
...
```

Caution

The runtime plusargs +dumpports+ieee and +dumpports+lsi have priority over this routine.

The format of files created by calls to the \$dumpports and \$lsi_dumpports tasks are not affected by this routine.

acc_lsi_dumpports_vhdl_enable

Syntax

```
void acc_lsi_dumpports_vhdl_enable(int enable)
```

The valid enable integer parameters are as follows:

1 enables VHDL drivers

0 disables VHDL drivers

Synopsis

Use this routine to enable or disable the inclusion of VHDL drivers in the determination of driver values.

Returns

No return value.

Example E-20 Example of acc_lsi_dumpports_vhdl_enable

```
#include "acc_user.h"
#include "vcs_acc_user.h"

handle instance = acc_handle_by_name("test_bench.device",
```

```

0) ;
char *outfile1 = "device.evcd1";
char *outfile2 = "device.evcd2";

/* Include VHDL drivers in this report */
acc_lsi_dumports_vhdl_enable(1);
acc_lsi_dumports_call(instance, outfile1);

/* Exclude VHDL drivers from this report */
acc_lsi_dumports_vhdl_enable(0);
acc_lsi_dumports_call(instance, outfile1);

...

```

Caution

This routine has precedence over the `+dumports+vhdl+enable` and `+dumports+vhdl+disable` runtime options.

Access Routines for Line Callbacks

VCS includes a number of access routines to monitor code execution. These access routines are as follows:

`acc_mod_lcb_add`

Registers a line callback routine with a module so that VCS calls the routine whenever VCS executes the specified module. See “[“acc_mod_lcb_add” on page 60](#) for details.

`acc_mod_lcb_del`

Unregisters a line callback routine previously registered with the `acc_mod_lcb_add()` routine. See “[“acc_mod_lcb_del” on page 63](#) for details.

`acc_mod_lcb_enabled`

Tests to see if line callbacks is enabled. See “[“acc_mod_lcb_enabled” on page 65](#) for details.

`acc_mod_lcb_fetch`

Returns an array of breakable lines. See “[“acc_mod_lcb_fetch” on page 66](#) for details.

`acc_mod_lcb_fetch2`

Returns an array of breakable lines. See “[“acc_mod_lcb_fetch2” on page 67](#) for details.

`acc_mod_sfi_fetch`

Returns the source file composition for a module. See “[“acc_mod_sfi_fetch” on page 69](#) for details.

acc_mod_lcb_add

Syntax

```
void acc_mod_lcb_add(handle handleModule,  
                      void (*consumer)(), char *user_data)
```

Synopsis

Registers a line callback routine with a module so that VCS calls the routine whenever VCS executes the specified module.

The prototype for the callback routine is:

```
void consumer(char *filename, int lineno, char *user_data,  
              int tag)
```

The tag field is a unique identifier that you use to distinguish between multiple `include files.

Protected modules cannot be registered for callback. This routine will just ignore the request.

Returns

No return value.

Example E-21 Example of acc_mod_lcb_add

```
#include <stdio.h>
#include "acc_user.h"
#include "vcs_acc_user.h"

/* VCS callback rtn */
void line_call_back(filename, lineno, userdata, tag)
char *filename;
int lineno;
char *userdata;
int tag;
{
    handle handle_mod = (handle)userdata;

    io_printf("Tag %2d, file %s, line %2d, module %s\n",
              tag, filename, lineno,
              acc_fetch_fullname(handle_mod));
}

/* register all modules for line callback (recursive) */
void register_lcb (parent_mod)
handle parent_mod;
{
    handle child = NULL;

    if (! acc_object_of_type(parent_mod, accModule)) return;

    io_printf("Registering %s\n",
              acc_fetch_fullname (parent_mod));

    acc_mod_lcb_add (parent_mod, line_call_back, parent_mod);

    while ((child = acc_next_child (parent_mod, child))) {
        register_lcb (child);
    }
}
```

acc_mod_lcb_del

Syntax

```
void acc_mod_lcb_del(handle handleModule,  
                      void (*consumer)(), char *user_data)
```

Synopsis

Unregisters a line callback routine previously registered with the `acc_mod_lcb_add()` routine.

Returns

No return value.

Example E-22 Example of acc_mod_lcb_del

```
#include <stdio.h>
#include "acc_user.h"
#include "vcs_acc_user.h"

/* VCS 4.x callback rtn */
void line_call_back(filename, lineno, userdata, tag)
char *filename;
int lineno;
char *userdata;
int tag;
{
    handle handle_mod = (handle)userdata;

    io_printf("Tag %2d, file %s, line %2d, module %s\n",
              tag, filename, lineno,
              acc_fetch_fullname(handle_mod));
}

/* unregister all line callbacks (recursive) */
void unregister_lcb (parent_mod)
handle parent_mod;
{
    handle child = NULL;

    if (! acc_object_of_type(parent_mod, accModule)) return;

    io_printf("Unregistering %s\n",
              acc_fetch_fullname (parent_mod));

    acc_mod_lcb_del (parent_mod, line_call_back, parent_mod);

    while ((child = acc_next_child (parent_mod, child))) {
        register_lcb (child);
    }
}
```

Caution

The module handle, consumer routine, and user data arguments must match those supplied to the `acc_mod_lcb_add()` routine for a successful delete.

For example, using the result of a call such as `acc_fetch_name()` as the user data will fail, because that routine returns a different pointer each time it is called.

acc_mod_lcb_enabled

Syntax

```
int acc_mod_lcb_enabled()
```

Synopsis

Test to see if line callbacks is enabled.

By default, the extra code required to support line callbacks is not added to a simulation executable. You can use this routine to determine if line callbacks have been enabled.

Returns

Non-zero if line callbacks are enabled; 0 if not enabled.

Example E-23 Example of acc_mod_lcb_enabled

```
if (! acc_mod_lcb_enable) {
    tf_warning("Line callbacks not enabled. Please recompile
with -line.");
}
else {
    acc_mod_lcb_add ( ... );
    ...
}
```

acc_mod_lcb_fetch

Syntax

```
p_location acc_mod_lcb_fetch(handle handleModule)
```

Synopsis

Returns an array of breakable lines.

This routine returns all the lines in a module that you can set breakpoints on.

Returns

The return value is an array of line number, file name pairs.

Termination of the array is indicated by a NULL file name field. The calling routine is responsible for freeing the returned array.

```
typedef struct t_location {  
    int line_no;  
    char *filename;  
} s_location, *p_location;
```

Returns NULL if the module has no breakable lines or is source protected.

Example E-24 Example of acc_mod_lcb_fetch

```
#include <stdio.h>  
#include "acc_user.h"  
#include "vcs_acc_user.h"  
  
void ShowLines(handleModule)  
handle handleModule;  
{  
    p_location plocation;  
  
    if ((plocation = acc_mod_lcb_fetch(handleModule)) != NULL)
```

```

{
    int i;

    io_printf("%s:\n", acc_fetch_fullname(handleModule));

    for (i = 0; plocation[i].filename; i++) {
        io_printf(" [%s:%d]\n",
                  plocation[i].filename,
                  plocation[i].line_no);
    }
    acc_free(plocation);
}
}

```

acc_mod_lcb_fetch2

Syntax

p_location2 acc_mod_lcb_fetch2(handle handleModule)

Synopsis

Returns an array of breakable lines.

This routine returns all the lines in a module that you can set breakpoints on.

The tag field is a unique identifier used to distinguish `include files. For example, in the following Verilog module, the breakable lines in the first `include of the file sequential.code have a different tag than the breakable lines in the second `include. (The

tag numbers will match the `vcs_srcfile_info_t->SourceFileTag` field. See the `acc_mod_sfi_fetch()` routine for details.)

```
module x;
initial begin
    `include sequential.code
    `include sequential.code
end
endmodule
```

Returns

The return value is an array of location structures. Termination of the array is indicated by a NULL filename field. The calling routine is responsible for freeing the returned array.

```
typedef struct t_location2 {
    int line_no;
    char *filename;
    int tag;
} s_location2, *p_location2;
```

Returns NULL if the module has no breakable lines or is source protected.

Example E-25 Example of acc_mod_lcb_fetch2

```
#include <stdio.h>
#include "acc_user.h"
#include "vcs_acc_user.h"

void ShowLines2(handleModule)
handle handleModule;
{
    p_location2 plocation;

    if ((plocation = acc_mod_lcb_fetch2(handleModule)) !=
```

```

NULL)  {
    int i;

    io_printf("%s:\n", acc_fetch_fullname(handleModule));

    for (i = 0; plocation[i].filename; i++) {
        io_printf("  file %s, line %d, tag %d\n",
                  plocation[i].filename,
                  plocation[i].line_no,
                  plocation[i].tag);
    }
    acc_free(plocation);
}
}

```

acc_mod_sfi_fetch

Syntax

```
vcs_srcfile_info_p acc_mod_sfi_fetch(handle handleModule)
```

Synopsis

Returns the source file composition for a module. This composition is a file name with line numbers, or, if a module definition is in more than one file, it is an array of `vcs_srcfile_info_s` struct entries specifying all the file names and line numbers for the module definition.

Returns

The returned array is terminated by a NULL SourceFileName field.
The calling routine is responsible for freeing the returned array.

```
typedef struct vcs_srcfile_info_t {
    char *SourceFileName;
    int SourceFileTag;
    int StartLineNum;
    int EndLineNum;
} vcs_srcfile_info_s, *vcs_srcfile_info_p;
```

Returns NULL if the module is source protected.

Example E-26 Example of acc_mod_sfi_fetch

```
#include <stdio.h>
#include "acc_user.h"
#include "vcs_acc_user.h"

void print_info (mod)
handle mod;
{
    vcs_srcfile_info_p infoa;

    io_printf("Source Info for Module %s:\n",
              acc_fetch_fullname(mod));

    if ((infoa = acc_mod_sfi_fetch(mod)) != NULL) {
        int i;
        for (i = 0; infoa[i].SourceFileName != NULL; i++) {
            io_printf(" Tag %2d, StartLine %2d, ",
                      infoa[i].SourceFileTag,
                      infoa[i].StartLineNum);
            io_printf("EndLine %2d, SrcFile %s\n",
                      infoa[i].EndLineNum,
                      infoa[i].SourceFileName);
        }
        acc_free(infoa);
    }
}
```

Access Routines for Source Protection

The `enclib.o` file provides a set of access routines that you can use to create applications which directly produce encrypted Verilog source code. Encrypted code can only be decoded by the VCS compiler. There is no user-accessible decode routine.

Note that both Verilog and SDF code can be protected. VCS knows how to automatically decrypt both.

VCS provides the following routines to monitor the port activity of a device:

`vcsspClose`

This routine frees the memory allocated by `vcsspInitialize()`. See “[vcsSpClose](#)” on page 76 for details.

`vcsspEncodeOff`

This routine inserts a trailer section containing the `'endprotected` compiler directive into the output file. It also toggles the encryption flag to false so that subsequent calls to `vcsspWriteString()` and `vcsspWriteChar()` will NOT cause their data to be encrypted. See “[vcsSpEncodeOff](#)” on page 76 for details.

`vcsspEncodeOn`

This routine inserts a trailer section containing the `'protected` compiler directive into the output file. It also toggles the encryption flag to false so that subsequent calls to `vcsspWriteString()` and `vcsspWriteChar()` will have their data encrypted. See “[vcsSpEncodeOn](#)” on page 78 for details.

`vcsspEncoding`

This routine gets the current state of encoding. See “[vcsspEncoding](#)” on page 79 for details.

`vcsspGetFilePtr`

This routine just returns the value previously passed to the `vcsspSetFilePtr()` routine. See “[vcsspGetFilePtr](#)” on page 80 for details.

`vcsspInitialize`

Allocates a source protect object. See “[vcsspInitialize](#)” on page 81 for details.

`vcsspOvaDecodeLine`

Decrypts one line. See “[vcsspOvaDecodeLine](#)” on page 82 for details.

`vcsspOvaDisable`

Switches to regular encryption. See “[vcsspOvaDisable](#)” on page 83 for details.

`vcsspOvaEnable`

Enables the OpenVera assertions (OVA) encryption algorithm. Tells VCS’s encrypter to use the OVA IP algorithm. See “[vcsspOvaEnable](#)” on page 85 for details.

`vcsspSetDisplayMsgFlag`

Sets the DisplayMsg flag. See “[vcsspSetDisplayMsgFlag](#)” on page 86 for details.

`vcsSpSetFilePtr`

Specifies the output file stream. See “[“vcsSpSetFilePtr” on page 87](#) for details.

`vcsSpSetLibLicenseCode`

Sets the OEM license code. See “[“vcsSpSetLibLicenseCode” on page 88](#) for details.

`vcsSpSetPliProtectionFlag`

Sets the PLI protection flag. See “[“vcsSpSetPliProtectionFlag” on page 89](#) for details.

`vcsSpWriteChar`

Writes one character to the protected file. See “[“vcsSpWriteChar” on page 90](#) for details.

`vcsSpWriteString`

Writes a character string to the protected file. See “[“vcsSpWriteString” on page 91](#) for details.

[Example E-27](#) outlines the basic use of the source protection routines.

Example E-27 Using the Source Protection Routines

```
#include <stdio.h>
#include "enclib.h"
void demo_routine()
{
    char *filename = "protected.vp";
    int write_error = 0;
    vcsspStateID esp;
    FILE *fp;

    /* Initialization */

    if ((fp = fopen(filename, "w")) == NULL) {
        printf("Error: opening file %s\n", filename);
        exit(1);
    }

    if ((esp = vcsspInitialize()) == NULL) {
        printf("Error: Initializing src protection
```

```

routines.\n");
    printf("      Out Of Memory.\n");
    fclose(fp);
    exit(1);
}

vcsspSetFilePtr(esp, fp); /* tell rtns where to write */

/* Write output */

write_error += vcsspWriteString(esp,
                                "This text will *not* be encrypted.\n");

write_error += vcsspEncodeOn(esp);
write_error += vcsspWriteString(esp,
                                "This text *will* be encrypted.");
write_error += vcsspWriteChar(esp, '\n');

write_error += vcsspEncodeOff(esp);
write_error += vcsspWriteString(esp,
                                "This text will *not* be encrypted.\n");

/* Clean up */

write_error += fclose(fp);
vcsspClose(esp);

if (write_error) {
    printf("Error while writing to '%s'\n", filename);
}
}

```

Caution

If you are encrypting SDF or Verilog code that contains include directives, you must switch off encryption (`vcsspEncodeOff`), output the include directive and then switch encryption back on. This ensures that when the parser begins reading the included file, it is in a known (non-decode) state.

If the file being included has proprietary data it can be encrypted separately. (Don't forget to change the `include compiler directive to point to the new encrypted name.)

vcsSpClose

Syntax

```
void vcsSpClose(vcsSpStateID esp)
```

Synopsis

This routine frees the memory allocated by `vcsSpInitialize()`. Call it when source encryption is finished on the specified stream.

Returns

No return value.

Example E-28 Example of vcsSpClose

```
vcsSpStateID esp = vcsSpInitialize();
...
vcsSpClose(esp);
```

vcsSpEncodeOff

Syntax

```
int vcsSpEncodeOff(vcsSpStateID esp)
```

Synopsis

This function performs two operations:

1. It inserts a trailer section that contains some closing information used by the decryption algorithm into the output file. It also inserts the `endprotected compiler directive in the trailer section.

- It toggles the encryption flag to false so that subsequent calls to `vcsspWriteString()` and `vcsspWriteChar()` will NOT cause their data to be encrypted.

Returns

Non-zero if there was an error writing to the output file, 0 if successful.

Example E-29 Example of vcsspEncodeOff

```

vcsspStateID esp = vcsspInitialize();
FILE *fp = fopen("protected.file", "w");
int write_error = 0;
if (fp == NULL) exit(1);

vcsspSetFilePtr(esp, fp);

if (vcsspWriteString(esp, "This text will *not* be encrypted.

++write_error;

if (vcsspEncodeOn(esp)) ++write_error;
if (vcsspWriteString(esp, "This text *will* be encrypted.

++write_error;

if (vcsspEncodeOff(esp)) ++write_error;
if (vcsspWriteString(esp, "This text will *not* be encrypted.

++write_error;

fclose(fp);
vcsspClose(esp);

```

Caution

You must call `vcsspInitialize()` and `vcsspSetFilePtr()` before calling this routine.

vcsSpEncodeOn

Syntax

```
int vcsSpEncodeOn(vcsspStateID esp)
```

Synopsis

This function performs two operations:

1. It inserts a header section which contains the 'protected' compiler directive into the output file. It also inserts some initial header information used by the decryption algorithm.
2. It toggles the encryption flag to true so that subsequent calls to `vcsspWriteString()` and `vcsspWriteChar()` will have their data encrypted.

Returns

Non-zero if there was an error writing to the output file, 0 if successful.

Example E-30 Example of vcsSpEncodeOn

```
vcsspStateID esp = vcsspInitialize();
FILE *fp = fopen("protected.file", "w");
int write_error = 0;

if (fp == NULL) exit(1);

vcsspSetFilePtr(esp, fp);

if (vcsspWriteString(esp, "This text will *not* be
encrypted.\n"))
    ++write_error;

if (vcsspEncodeOn(esp)) ++write_error;
if (vcsspWriteString(esp, "This text *will* be
```

```
encrypted.\n" )  
    ++write_error;  
  
if (vcsspEncodeOff(esp)) ++write_error;  
if (vcsspWriteString(esp, "This text will *not* be  
encrypted.\n" ))  
    ++write_error;  
fclose(fp);  
vcsspClose(esp);
```

Caution

You must call `vcsspInitialize()` and `vcsspSetFilePtr()` before calling this routine.

vcsspEncoding

Syntax

```
int vcsspEncoding(vcsspStateID esp)
```

Synopsis

Calling `vcsspEncodeOn()` and `vcsspEncodeOff()` turns encoding on and off. Use this function to get the current state of encoding.

Returns

1 for on, 0 for off.

Example E-31 Example of vcsSpEncoding

```
vcsspStateID esp = vcsspInitialize();
FILE *fp = fopen("protected.file", "w");

if (fp == NULL) { printf("ERROR: file ..."); exit(1); }

vcsspSetFilePtr(esp, fp);
...

if (! vcsspEncoding(esp))
    vcsspEncodeOn(esp)
...

if (vcsspEncoding(esp))
    vcsspEncodeOff(esp);

fclose(fp);
vcsspClose(esp);
```

vcsspGetFilePtr

Syntax

```
FILE *vcsspGetFilePtr(vcsspStateID esp)
```

Synopsis

This routine just returns the value previously passed to the `vcsspSetFilePtr()` routine.

Returns

File pointer or NULL if not set.

Example E-32 Example of vcsSpGetFilePtr

```
vcsspStateID esp = vcsspInitialize();
FILE *fp = fopen("protected.file", "w");
if (fp != NULL)
    vcsspSetFilePtr(esp, fp);
else
    /* doh! */

...
if ((gfp = vcsspGetFilePtr(esp)) != NULL) {
    /* Add comment before starting encryption */
    fprintf(gfp, "\n// TechStuff Version 2.2\n");
    vcsspEncodeOn(esp);
}
```

Caution

Don't use non-vcssp* routines (like `fprintf`) in conjunction with vcssp* routines, while encoding is enabled.

vcsspInitialize

Syntax

```
vcsspStateID vcsspInitialize(void)
```

Synopsis

This routine allocates a source protect object.

Returns a handle to a malloc'd object which must be passed to all the other source protection routines.

This object stores the state of the encryption in progress. When the encryption is complete, this object should be passed to `vcsspClose()` to free the allocated memory.

If you need to write to multiple streams at the same time (perhaps you're creating include or SDF files in parallel with model files), you can make multiple calls to this routine and assign a different file pointer to each handle returned.

Each call mallocs less than 100 bytes of memory.

Returns

The `vcsspStateID` pointer or NULL if memory could not be malloc'd.

Example E-33 Example of vcsspStateID

```
vcsspStateID esp = vcsspInitialize();
if (esp == NULL) {
    fprintf(stderr, "out of memory\n");
    ...
}
```

Caution

This routine must be called before any other source protection routine.

A NULL return value means the call to `malloc()` failed. Your program should test for this.

vcsspOvaDecodeLine

Syntax

```
vcsspStateID vcsspOvaDecodeLine(vcsspStateID esp, char
*line)
```

Synopsis

This routine decrypts one line.

Use this routine to decrypt one line of protected IP code such as OVA code. Pass in a null vcsspStateID handle with the first line of code and a non-null handle with subsequent lines.

Returns

Returns NULL when the last line has been decrypted.

Example E-34 Example of vcsSpOvaDecodeLine

```
#include "enclib.h"

if (strcmp(linebuf, "'protected_ip synopsys\n") == 0) {
    /* start IP decryption */
    vcsspStateID esp = NULL;
    while (fgets(linebuf, sizeof(linebuf), infile)) {
        /* linebuf contains encrypted source */
        esp = vcsSpOvaDecodeLine(esp, linebuf);
        if (linebuf[0]) {
            /* linebuf contains decrypted source */
            ...
        }
        if (!esp) break; /* done */
    }
    /* next line should be 'endprotected_ip */
    fgets(linebuf, sizeof(linebuf), infile);
    if (strcmp(linebuf, "'endprotected_ip\n") != 0) {
        printf("warning - expected 'endprotected_ip\n");
    }
}
```

vcsSpOvaDisable

Syntax

```
void vcsSpOvaDisable(vcsspStateID esp)
```

Synopsis

This routine switches to regular encryption. It tells VCS's encrypter to use the standard algorithm. This is the default mode.

Returns

No return value.

Example E-35 Example of vcsSpOvaDisable

```
#include "enclib.h"
#include "encint.h"

int write_error = 0;
vcsSpStateID esp;

if ((esp = vcsSpInitialize()) == NULL) printf("Out Of Memory");

vcsSpSetFilePtr(esp, fp); /* previously opened FILE* pointer */

/* Configure for OVA IP encryption */
vcsSpOvaEnable(esp, "synopsys");

if (vcsSpWriteString(esp, "This text will NOT be
encrypted.\n"))
    ++write_error;

if (vcsSpEncodeOn(esp)) ++write_error;

if (vcsSpWriteString(esp, "This text WILL be encrypted.\n"))
    ++write_error;

if (vcsSpEncodeOff(esp)) ++write_error;

if (vcsSpWriteString(esp, "This text will NOT be
```

```

        encrypted.\n" ) )
        ++write_error;
/* Switch back to regular encryption */
vcsSpOvaDisable(esp);

if (vcsSpEncodeOn(esp)) ++write_error;

if (vcsSpWriteString(esp, "This text WILL be encrypted.\n" ))
    ++write_error;

if (vcsSpEncodeOff(esp)) ++write_error;

vcsSpClose(esp);

```

vcsSpOvaEnable

Syntax

```
void vcsSpOvaEnable(vcsSpStateID esp, char *vendor_id)
```

Synopsis

Enables the OpenVera assertions (OVA) encryption algorithm. Tells VCS's encrypter to use the OVA IP algorithm.

Returns

No return value.

Example E-36 Example of vcsSpOvaEnable

```

#include "enclib.h"
#include "encint.h"

int write_error = 0;
vcsSpStateID esp;

if ((esp = vcsSpInitialize()) == NULL) printf("Out Of Memory");

vcsSpSetFilePtr(esp, fp); /* previously opened FILE* pointer

```

```

*/



/* Configure for OVA IP encryption */
vcsSpOvaEnable(esp, "synopsys");

if (vcsspWriteString(esp, "This text will NOT be
encrypted.\n"))
    ++write_error;

if (vcsspEncodeOn(esp)) ++write_error;

if (vcsspWriteString(esp, "This text WILL be encrypted.\n"))
    ++write_error;

if (vcsspEncodeOff(esp)) ++write_error;

if (vcsspWriteString(esp, "This text will NOT be
encrypted.\n"))
    ++write_error;
/* Switch back to regular encryption */
vcsspOvaDisable(esp);

if (vcsspEncodeOn(esp)) ++write_error;

if (vcsspWriteString(esp, "This text WILL be encrypted.\n"))
    ++write_error;

if (vcsspEncodeOff(esp)) ++write_error;

vcsspClose(esp);

```

vcsspSetDisplayMsgFlag

Syntax

```
void vcsspSetDisplayMsgFlag(vcsspStateID esp, int enable)
```

Synopsis

This routine sets the DisplayMsg flag. By default, the VCS compiler does not display decrypted source code in its error or warning messages. Use this routine to enable this display.

Returns

No return value.

Example E-37 Example of vcsSpSetDisplayMsgFlag

```
vcsspStateID esp = vcsspInitialize();
vcsspSetDisplayMsgFlag(esp, 0);
```

vcsspSetFilePtr

Syntax

```
void vcsspSetFilePtr(vcsspStateID esp, FILE *fp)
```

Synopsis

This routine specifies the output file stream. Before using the `vcsspWriteChar()` or `vcsspWriteString()` routines, you must specify the output file stream.

Returns

No return value.

Example E-38 Example of vcsspSetFilePtr

```
vcsspStateID esp = vcsspInitialize();
FILE *fp = fopen("protected.file", "w");
if (fp != NULL)
    vcsspSetFilePtr(esp, fp);
else
    /* abort */
```

vcsSpSetLibLicenseCode

Syntax

```
void vcsspSetLibLicenseCode(vcsspStateID esp, unsigned int  
    code)
```

Synopsis

This routine sets the OEM library license code that will be added to each protected region started by `vcsSpEncodeOn()`.

This code can be used to protect library models from unauthorized use.

When the VCS parser decrypts the protected region, it verifies that the end user has the specified license. If the license does not exist or has expired, VCS exits.

Returns

No return value.

Example E-39 Example of vcsSpSetLibLicenseCode

```
unsigned int lic_code = MY_LICENSE_CODE;  
vcsspStateID esp = vcsspInitialize();  
...  
  
/* The following text will be encrypted and licensed */  
vcsspSetLibLicenseCode(esp, code); /* set license code */  
vcsspEncodeOn(esp); /* start protected region */  
vcsspWriteString(esp, "this text will be encrypted and  
licensed");  
vcsspEncodeOff(esp); /* end protected region */  
  
/* The following text will be encrypted but unlicensed */  
vcsspSetLibLicenseCode(esp, 0); /* clear license code */  
vcsspEncodeOn(esp); /* start protected region */  
vcsspWriteString(esp, "this text encrypted but not
```

```
licensed");
vcsSpEncodeOff(esp);                                /* end protected region */
```

Caution

The rules for mixing licensed and unlicensed code is determined by your OEM licensing agreement with Synopsys.

The code segment in [Example E-39](#) shows how to enable and disable the addition of the license code to the protected regions. Normally you would call this routine once, that is, after calling `vcsSpInitialize()` and before the first call to `vcsSpEncodeOn()`.

vcsSpSetPliProtectionFlag

Syntax

```
void vcsSpSetPliProtectionFlag(vcsSpStateID esp, int
enable)
```

Synopsis

This routine sets the PLI protection flag. You can use it to disable the normal PLI protection that is placed on encrypted modules. The output files will still be encrypted, but CLI and PLI users will not be prevented from accessing data in the modules.

This routine only affects encrypted Verilog files. Encrypted SDF files, for example, are not affected.

Returns

No return value.

Example E-40 Example of vcsSpSetPliProtectionFlag

```
vcsspStateID esp = vcsspInitialize();
vcsspSetPliProtectionFlag(esp, 0); /* disable PLI protection
*/
```

Caution

Turning off PLI protection will allow users of your modules to access object names, values, etc. In essence, the source code for your module could be substantially reconstructed using the CLI commands and ACC routines.

vcsspWriteChar

Syntax

```
void vcsspSetPliProtectionFlag(vcsspStateID esp, int
enable)
```

Synopsis

This routine writes one character to the protected file.

If encoding is enabled (see “[vcsspEncodeOn](#)” on page 78) the specified character is encrypted as it is written to the output file.

If encoding is disabled (see “[vcsspEncodeOff](#)” on page 76) the specified character is written as-is to the output file (no encryption.)

Returns

Non-zero if the file pointer has not been set (see “[vcsspSetFilePtr](#)” on page 87) or if there was an error writing to the output file (out-of-disk-space, etc.)

Returns 0 if the write was successful.

Example E-41 Example of vcsSpWriteChar

```
vcsspStateID esp = vcsspInitialize();
FILE *fp = fopen("protected.file", "w");
int write_error = 0;

if (fp == NULL) exit(1);

vcsspSetFilePtr(esp, fp);

if (vcsspWriteChar(esp, 'a')) /* This char will *not* be
encrypted.*/
    ++write_error;

if (vcsspEncodeOn(esp))
    ++write_error;

if (vcsspWriteChar(esp, 'b')) /* This char *will* be
encrypted. */
    ++write_error;
if (vcsspEncodeOff(esp))
    ++write_error;

fclose(fp);
vcsspClose(esp);
```

Caution

vcsspInitialize() and vcsspSetFilePtr() must be called prior to calling this routine.

vcsspWriteString

Syntax

```
int vcsspWriteString(vcsspStateID esp, char *s)
```

Synopsis

This routine writes a character string to the protected file.

If encoding is enabled (see “[vcsSpEncodeOn](#)” on page 78) the specified string is encrypted as it is written to the output file.

If encoding is disabled (see “[vcsSpEncodeOff](#)” on page 76) the specified string will be written as-is to the output file (no encryption.)

Returns

Non-zero if the file pointer has not been set (see “[vcsSpSetFilePtr](#)” on page 87) or if there was an error writing to the output file (out-of-disk-space, etc.)

Returns 0 if the write was successful.

Example E-42 Example of vcsSpWriteString

```
vcsspstateid esp = vcsspinitialize();
FILE *fp = fopen("protected.file", "w");
int write_error = 0;

if (fp == NULL) exit(1);

vcsspsetfileptr(esp, fp);

if (vcsspwritestring(esp, "This text will *not* be
encrypted.\n"))
    ++write_error;

if (vcsspencodeon(esp)) ++write_error;
if (vcsspwritestring(esp, "This text *will* be
encrypted.\n"))
    ++write_error;

if (vcsspencodeoff(esp)) ++write_error;
if (vcsspwritestring(esp, "This text will *not* be
```

```
encrypted.\n" )  
++write_error;  
  
fclose(fp);  
vcsspClose(esp);
```

Caution

vcsspInitialize() and vcsspSetFilePtr() must be called prior to calling this routine.

Access Routine for Signal in a Generate Block

There is only one access routine for signals in generate blocks.

acc_object_of_type

Syntax

```
bool acc_object_of_type(accGenerated, sigHandle)
```

Synopsis

This routine returns true if the signal is in a generate block.

Returns

1 - if the signal is in a generate block.

0 - if the signal is not in a generate block.

VCS API Routines

Typically VCS controls the PLI application. If you write your application so that it controls VCS, you need these API routines.

Vcsinit()

When VCS is run in slave mode, you can call this function to elaborate the design and to initialize various data structures, scheduling queues, etc. that VCS uses. After this routine executes, all the initial time 0 events, such as the execution of initial blocks, are scheduled.

Call the `vmc_main(int argc, char *argv)` routine to pass runtime flags to VCS before you call `VcsInit()`.

VcsSimUntil()

This routine tells VCS to schedule a stop event at the specified simulation time and execute all scheduled simulation events until it executes the stop event. The syntax for this routine is as follows:

```
VcsSimUntil (unsigned int* t)
```

Argument `t` is for specifying the simulation time. It needs two words. The first [0] is for simulation times from 0 to $2^{32} - 1$, the second is for simulation times that follow.

If any events are scheduled to occur after time `t`, their execution must wait for another call to `VcsSimUntil`.

If τ is less than the current simulation time, VCS returns control to the calling routine.

Index

Symbols

- ams_discipline B-51
- ams_iereport B-52
- assert 2-7, B-10, B-28
- B B-62
- C B-48
- c B-46
- CC B-46
- cc B-46
- CFLAGS B-47
- cm_fsmopt allowTemp B-24
- cm_fsmopt optimist B-24
- cm_fsmopt report2StateFsms B-24
- cm_fsmopt reportvalues B-24
- cm_fsmopt reportWait B-24
- cm_fsmopt reportXassign B-24
- cm_fsmresetfilter B-25
- cm_line contassign B-25
- cm_scope B-27
- cpp B-47
- debug 2-6, B-27
- debug_all 2-6, B-28
- doc 2-3, B-9
- e name_for_main B-38
- E program runtime option C-24
- f filename B-35
- gui 2-7, B-28
- h 2-3, B-9
- help 2-3, B-9
- ID 2-3, B-45
- jnumber_of_CPUs B-48
- I 2-13, C-17
- I filename 2-9, 2-14, B-61, C-15
- ld linker B-46
- LDFLAGS B-46
- lmc-swift B-40
- lmc-swift-template B-41
- lname B-46
- load 23-30, B-38
- Mdirectory B-8
- negdelay B-34
- nolncComp B-8
- ntb B-13
- ntb_opts B-14
- ntb_sfname B-16
- ntb_vipext B-17
- ntb_vl B-17
- o name 2-9, B-62
- O number B-49
- O0 B-48
- override_timescale B-54
- P pli.tab B-38
- parameters 2-8, 4-6, B-52
- platform B-62
- PP D-19

-pvalue 2-7, 4-5, B-52
 -q 2-8, B-42, C-15
 -R 2-7, B-19
 -sysc B-58
 -u B-61
 -ucli 2-13, B-28, C-17
 -V 2-8, B-42, C-16
 -v 2-3, B-4
 -vcf filename C-19
 -Vt B-42
 -Xman B-50
 -Xmangle B-50
 -Xnoman B-51
 -Xnomangle B-51
 -y 2-4, B-4
 "A" specifier of abstract access 23-41
 "C" specifier of direct access 23-41
 \$assert_monitor 20-9, D-26
 \$assert_monitor_off 20-9, D-27
 \$assert_monitor_on 20-9, D-27
 \$assertkill D-11
 \$assertoff D-11
 \$asserton D-11
 \$async\$and\$array D-37
 \$bitstoreal D-28
 \$countdrivers D-41
 \$deposit D-41
 \$disable_warnings D-34
 \$display D-29
 \$dist_exponential D-39
 \$dist_normal D-39
 \$dist_poisson D-39
 \$dist_uniform D-39
 \$dumpall D-12
 \$dumpfile D-13
 \$dumpflush D-13
 \$dumplimit D-13
 \$dumpoff D-13
 \$dumpon D-13
 \$dumports 7-23, D-16
 \$dumportsall D-17
 \$dumpportsflush D-18
 \$dumpportslimit D-18
 \$dumpportsoff D-17
 \$dumpportson D-17
 \$dumpvars D-13
 \$enable_warnings D-35
 \$error D-11
 \$fatal D-10
 \$fclose D-30
 \$fdisplay D-30
 \$ferror D-30
 \$fflush D-14, D-30
 \$fflushall D-14
 \$fgetc D-30
 \$fgets D-30
 \$finish D-34
 \$fmonitor D-30
 \$fopen D-30
 \$fread D-31
 \$fscanf D-31
 \$fseek D-31
 \$fstobe D-31
 \$ftell D-31
 \$fwrite D-31
 \$getpattern D-41
 \$gr_waves D-14
 \$hold D-35
 \$info D-11
 \$itor D-28
 \$log D-28
 \$lsl_dumports 3-38–3-44, 7-22, D-15
 \$monitor D-29
 \$monitoroff D-29
 \$monitoron D-29
 \$nolog D-28
 \$period D-35
 \$printtimescale D-33
 \$q_add D-37
 \$q_exam D-38
 \$q_full D-38
 \$q_initialize D-38

\$q_remove D-38
 \$random 3-37, D-39
 \$readmemb D-32
 \$readmemh D-32
 \$realtime D-38
 \$realtobits D-28
 \$recovery D-35
 \$recrem D-36
 \$removal D-36
 \$reset D-39
 \$reset_count D-40
 \$reset_value D-40
 \$restart D-42
 \$rtoi D-29
 \$save D-41
 \$sdf_annotate D-40
 \$setup D-36
 \$setuphold D-37
 \$skew D-37
 \$sreadmemb D-32
 \$sreadmemh D-32
 \$stime D-38
 \$stop D-33
 \$strobe D-29
 \$sync\$nor\$plane D-37
 \$system D-27
 \$systemf D-27
 \$test\$plusargs D-40
 \$time D-38
 \$timeformat D-33
 \$ungetc D-32
 \$uniq_prior_checkoff system task 14-25
 \$uniq_prior_checkon system task 14-25
 \$value\$plusargs 5-5
 \$vcplusplusdeltacycleoff 7-21
 \$vcplusplusdeltacycleon 7-20
 \$vcplusplusmemoff 7-9
 \$vcplusplusmemon 7-9
 \$vcplusplusmemorydump 7-9
 \$warning D-11
 \$width D-37
 \$write D-29
 \$writememb D-33
 \$writememh D-33
 %CELL 23-14, 23-16
 %TASK 23-14
 +abstract 23-121
 +acc+level_number 23-20, B-37
 +ad B-51
 +allhdrs 23-121
 +allmtm B-29
 +applylearn 23-24–23-28
 +auto2protect B-49
 +auto3protect B-49
 +autoprotect B-49
 +charge_decay B-29
 +define+macro=value 2-9, B-61
 +delay_mode_distributed 11-24, B-30
 +delay_mode_path 11-23, B-29
 +delay_mode_unit 11-24, B-30
 +delay_mode_zero 11-24, B-29
 +deleteprotected B-50
 +inmdir 2-5, B-6
 +libext 2-5, B-6
 +liborder 2-5, B-6
 +librescan B-6
 +libverbose B-7, B-41
 +lint B-41
 +list 23-121
 +maxdelays B-30, C-20
 +memcbk B-60
 +mindelays B-30, C-21
 +multisource_int_delays 11-9, B-30
 +nbaopt B-31
 +neg_tchk 11-43, 11-51, B-34
 +no_notifier 11-44, B-33, C-14
 +no_pulse_msg C-16
 +no_tchk_msg 11-44, B-33, C-14
 +nocelldefinepli+0 B-44
 +nocelldefinepli+1 B-44
 +nocelldefinepli+2 B-44
 +noerrorIOPCWM B-59

+nolibcell B-44
 +nospecify 11-44, B-32
 +notimingcheck 11-44, B-32, C-14
 +ntb_cache_dir C-3
 +ntb_debug_on_error C-3
 +ntb_enable_solver_trace C-3
 +ntb_enable_solver_trace_on_failure C-3
 +ntb_enable_solver_trace_on_failure=value C-3
 +ntb_enable_solver_trace C-3
 +ntb_exit_on_error C-4
 +ntb_load C-4
 +ntb_random_seed C-4
 +ntb_random_seed_automatic C-4
 +ntb_solver_mode C-5
 +ntb_solver_mode=value C-5
 +ntb_stop_on_error C-5
 +NTC2 11-50, B-35
 +old_ntc B-35
 +optconfigfile 10-6, B-19
 +overlap 11-54, B-35
 +override_model_delays C-25
 +pathpulse B-32
 +pli_unprotected B-50
 +plusarg_ignore B-37
 +plusarg_save B-36
 +plus-options C-25
 +prof B-35
 +protect_file_suffix B-50
 +pulse_e/number 11-10, 11-12, 11-14, 11-19, 11-20, B-33
 +pulse_int_e 11-9, 11-10, 11-12, 11-14, B-34
 +pulse_int_r 11-9, 11-10, 11-12, 11-14, B-33
 +pulse_on_detect 11-20, B-34
 +pulse_on_event 11-19, B-34
 +pulse_r/number 11-10, 11-12, 11-14, 11-19, 11-20, B-33
 +putprotect+target_dir B-50
 +rad 10-6, B-18
 +sdf_nocheck_celltype B-31
 +sdfprotect_file_suffix B-50
 +sdfverbose C-16
 +systemverilogext B-17
 +timopt 11-26
 +transport_int_delays 11-9, 11-12, 11-14, B-31
 +transport_path_delays 11-9, 11-12, 11-14, B-31
 +typdelays B-30, C-22
 +v2k B-58
 +vc 23-120, B-39
 +vcdfile B-28
 +vcs+boundscheck B-59
 +vcs+dumpoff+t+ht C-20
 +vcs+dumpon+t+ht C-20
 +vcs+finish 5-24, C-15
 +vcs+flush+all C-23
 +vcs+flush+dump C-23
 +vcs+flush+fopen C-23
 +vcs+flush+log C-23
 +vcs+ignorestop C-25
 +vcs+initmem B-18
 +vcs+initreg B-18
 +vcs+learn+pli 23-24–23-28, C-24
 +vcs+lic+vcsi B-45, C-23
 +vcs+lic+wait 2-3, B-45, C-24
 +vcs+mipd+noalias C-26
 +vcs+nostdout C-16
 +vcs+saif_libcell 24-2
 +vcs+stop 5-24, C-15
 +vcsi+lic+vcs B-45, C-23
 +vcsi+lic+wait B-45
 +vera_stop_on_error C-5
 +verilog1995ext B-18
 +verilog2001ext B-18
 +vpdffile B-28
 +vpdffileswitchsize B-28
 +vpi B-38
 +warn 3-38, B-43
 'celldefine D-2, D-3
 'default_nettype D-2
 'define D-3

'delay_mode_distributed D-5
 'delay_mode_path D-5
 'delay_mode_unit D-5
 'delay_mode_zero D-5
 'else D-3
 'elseif D-3
 'endcelldefine D-2
 'endif D-3
 'endprotect D-7
 'endprotected D-7
 'ifdef D-4
 'include D-8
 'line D-10
 'noportcoerce 3-16, D-7
 'nounconnected_drive D-10
 'portcoerce D-7
 'protect D-7
 'protected D-7
 'resetall D-3
 'timescale D-8
 'unconnected_drive D-10
 'undef D-5
 'uselib D-9
 'vcs_mipdexpand D-6

mipb 23-13
 mp 23-13
 prx 23-12
 r 23-12, 23-17
 rw 23-12, 23-17
 s 23-12
 specifying 23-10–23-19
 tchk 23-13
 access routines for abstract access of C/C++ functions 23-66–23-115
 +ad B-51
 +allhdrs 23-121
 +allmtm B-29
 -ams_discipline B-51
 -ams_iereport B-52
 aop
 advice
 before/after/around 15-17
 dominates 15-8
 extends directive 15-4
 placement element
 after 15-12
 around 15-12
 +applylearn 23-24–23-28
 arb.v 13-5
 args PLI Specification 23-8
 array
 output and inout argument type 23-57
 -assert 2-7, B-10, B-28, C-5
 \$assert_monitor 20-9, D-26
 \$assert_monitor_off 20-9, D-27
 \$assert_monitor_on 20-9, D-27
 Assertion Failure Summary Pane 8-53
 Assertion failures 8-53
 \$assertkill D-11
 \$assertoff D-11
 \$asserton D-11
 \$async\$and\$array D-37
 +auto2protect B-49
 +auto3protect B-49
 +autoproTECT B-49

A

"A" specifier of abstract access 23-41
 +abstract 23-121
 abstract access for C/C++ functions
 access routines for 23-66–23-115
 enabling with a compile-time option 23-121
 using 23-64–23-115
 +acc+level_number 23-20, B-37
 ACC capabilities 23-27
 cbk 23-12, 23-18
 cbka 23-12
 frc 23-12, 23-18
 gate 23-13
 mip 23-13

B

-B B-62
binary radix 8-25
bit
 C/C++ function argument type 23-43
 C/C++ function return type 23-42
 input argument type 23-56
 output and inout argument type 23-56
 reg data type in two-state simulation 23-38
\$bitstoreal D-28

C

-C B-48
-c 13-5, B-46
C code generating
 halt before compiling the generated C code
 B-48
 passing options to the compiler B-46
 specifying another compiler B-46
 suppressing optimization for faster
 compilation B-48
C compiler, environment variable specifying
the A-4
"C" specifier of direct access 23-41
C/C++ functions
 argument direction 23-41, 23-43
 argument type 23-42, 23-43
 calling 23-46–23-48
 declaring 23-40–23-46
 extern declaration 23-40
 in a Verilog environment 23-38–23-39
 return range 23-41
 return type 23-41, 23-42
 using abstract access 23-64–23-115
 access routines for 23-66–23-115
 using direct access 23-54–23-64
 examples 23-57–23-61
C1 cursor 8-25
call PLI specification 23-7
calling C/C++ functions in your Verilog code
23-46–23-48
cbk ACC capability 23-12, 23-18

cbka ACC capability 23-12
-CC B-46
-cc B-46
'celldefine D-2, D-3
-CFLAGS B-47
char*
 direct access for C/C++ functions
 formal parameter type 23-54
char**
 direct access for C/C++ functions
 formal parameter type 23-54
+charge_decay B-29
check argument to -ntb_opts B-14
check PLI specification 23-7
check=all B-15
check=fixed B-15
clock signals 11-25–11-30
-cm 2-9, 2-12, B-19, C-11
-cm_assert_dir C-11
-cm_assert_hier B-21
-cm_cond B-21
-cm_constfile B-23
-cm_dir B-23, C-12
-cm_fsmcfg B-23
-cm_fsmopt B-23
-cm_fsmopt allowTmp B-24
-cm_fsmopt optimist B-24
-cm_fsmopt report2StateFsms B-24
-cm_fsmopt reportvalues B-24
-cm_fsmopt reportWait B-24
-cm_fsmopt reportXassign B-24
-cm_fsmresetfilter B-25
-cm_glitch C-13
-cm_hier B-25
-cm_ignore pragmas B-25
-cm_libs B-25
-cm_line contassign B-25
-cm_log C-13
-cm_name B-26, C-13
-cm_noconst B-26
-cm_pp B-26

-cm_scope B-27
 -cm_tgl B-27
 compiler directives D-1–D-10
 compile-time options B-1–B-62
 -ntb_cmp B-14
 compiling
 incremental compilation
 triggering ??–10-3
 verbose messages 2-8, B-42
 \$countdrivers D-41
 coverage
 coverage_load() 13-32, 14-6
 single coverage_group 14-6
 loading coverage data
 coverage_instance() 13-33, 14-7
 loading embedded coverage data
 coverage_instance() 13-34, 14-8
 coverage_instance() 14-7
 coverage_load 13-32, 14-6
 -cpp B-47
 Create Marker (CSM selection) 8-33
 cursor C1 8-31
 cursor C2 8-31

D

Data Pane 8-7
 data PLI specification 23-7
 Data Type Mapping File
 VCS/SystemC cosimulation interface 22-38
 -debug 2-6, B-27
 -debug_all 2-6, B-28
 declaring C/C++ functions in your Verilog code
 23-40–23-46
 ‘default_netttype D-2
 ‘define D-3
 +define+macro=value 2-9, B-61
 delay values
 back annotating to your design D-40
 +delay_mode_distributed 11-24, B-30
 ‘delay_mode_distributed D-5
 +delay_mode_path 11-23, B-29

‘delay_mode_path D-5
 +delay_mode_unit 11-24, B-30
 ‘delay_mode_unit D-5
 +delay_mode_zero 11-24, B-29
 ‘delay_mode_zero D-5
 +deleteprotected B-50
 Denali 30-1
 dep_check argument to -ntb_opts B-15
 \$deposit D-41
 Design Description 13-6
 direct access for C/C++ functions
 examples 23-57–23-61
 formal parameters
 types 23-54
 rules for parameter types 23-55–23-57
 using 23-54–23-120
 direction of a C/C++ function argument 23-43
 \$disable_warnings D-34
 \$display D-29
 DISPLAY_VCS_HOME A-3
 displaying
 scope data 8-13
 \$dist_exponential D-39
 \$dist_normal D-39
 \$dist_poisson D-39
 \$dist_uniform D-39
 dividers, signal 8-27
 DKI communication 22-6
 -do 2-14, C-17
 -doc 2-3, B-9
 double*
 direct access for C/C++ functions
 formal parameter type 23-54
 \$dumpall D-12
 \$dumpfile D-13
 \$dumpflush D-13
 \$dumplimit D-13
 \$dumpoff D-13
 \$dumpon D-13
 \$dumports 7-23, D-16
 \$dumportsall D-17

\$dumpportsflush D-18
\$dumpportslimit D-18
\$dumpportsoff D-17
\$dumpportson D-17
\$dumpvars D-13
duplicate signals, displaying 8-27
DVE
 starting 8-9
DVE version 8-2

E

-e name_for_main B-38
-E program C-24
'else D-3
'elseif D-3
\$enable_warnings D-35
enabling
 only where used in the last simulation 23-27
'endcelldefine D-2
'endif D-3
'endprotect D-7
'endprotected D-7
Environment variables 1-6–1-7, A-1–A-5
\$error D-11
event controls
 in condition coverage 12-5
extends directive
 advice 15-4
 introduction 15-4
extern declaration 23-40
extern declarations 23-62

F

-f filename B-35
\$fatal D-10
fclose D-30
\$fdisplay D-30
\$ferror D-30
\$fflush D-14, D-30

\$fflushall D-14
\$fgetc D-30
\$fgets D-30
-file 2-6, B-36
Files
 tokens.v B-50
\$finish D-34
\$fmonitor D-30
\$fopen D-30
four state Verilog data
 stored in vec32 23-49
frc ACC capability 23-12, 23-18
\$fread D-31
\$fscanf D-31
\$fseek D-31
\$fstobe D-31
\$ftell D-31
\$fwrite D-31

G

gate ACC capability 23-13
\$getpattern D-41
gmake A-2
\$gr_waves D-14
-gui 2-7, B-28
gui, option 8-3

H

-h 2-3, B-9
-help 2-3, B-9
hexadecimal radix 8-25
\$hold D-35

I

-ID 2-3, B-45
'ifdef D-4
-ignore B-9
Ignoring Calls and License Checking C-19

+inmdir 2-5, B-6
 'include D-8
 Incremental Compilation B-7–??
 \$info D-11
 -ignore B-9
 inout
 C/C++ function argument direction 23-43
 input
 C/C++ function argument direction 23-43
 inserting new markers 8-33
 int
 C/C++ function argument type 23-43
 C/C++ function return type 23-42
 direct access for C/C++ functions
 formal parameter type 23-54
 input argument type 23-56
 output and inout argument type 23-56
 int*
 direct access for C/C++ functions
 formal parameter type 23-54
 interface 13-7
 Interface Description 13-15
 interval between cursors 8-31
 invoking DVE 8-9
 \$itor D-28

J
 -jnumber_of_CPUs B-48

M
 Main Window
 example 8-5
 make A-2
 -Marchive B-7
 Markers dialog box 8-33, 8-34
 maxargs PLI specification 23-8
 +maxdelays B-30, C-20
 -Mdirectory B-8
 +memcbk B-60
 memories

N
 -I 2-13, C-17

O
 -O 2-13, C-17

P
 -P 2-13, C-17

R
 -R 2-13, C-17

S
 -S 2-13, C-17

T
 -T 2-13, C-17

U
 -U 2-13, C-17

V
 -V 2-13, C-17

W
 -W 2-13, C-17

X
 -X 2-13, C-17

Z
 -Z 2-13, C-17

- sparse memory models 3-28
- Memory Modeler - Advanced Verification (MMAV) 30-1
 - memory size limits 3-27
 - merging signal groups 8-18
 - minargs PLI specification 23-8
 - +mindelays B-30, C-21
 - mip ACC capability 23-13
 - mipb ACC capability 23-13
 - misc PLI specification 23-7
 - module description , Verilog 13-15
 - module path delays
 - disabling for an instance 11-25
 - suppressing B-32
 - in specific module instances 11-25
 - \$monitor D-29
 - \$monitoroff D-29
 - \$monitoron D-29
 - mp ACC capability 23-13
 - +multisource_int_delays 11-9, B-30

N

- Name column (signal pane) 8-25
- +nbaopt B-31
- +neg_tchk 11-43, 11-51, B-34
- negdelay B-34
- new markers, inserting 8-33
- no_file_by_file_pp argument to -ntb_opts B-15
- +no_identifier C-14
- +no_notifier 11-44, B-33
- +no_pulse_msg C-16
- +no_tchk_msg 11-44, B-33, C-14
- +nocelldefinepli+1 B-44
- nocelldefinepli PLI specification 23-8
- +nocelldefinepli+0 B-44
- +nocelldefinepli+2 B-44
- noIncrComp B-8
- +nolibcell B-44
- \$nolog D-28
- 'noportcoerce 3-16, D-7

- +nospecify 11-44, B-32
- notice 2-8, B-41
- +notimingcheck 11-44, B-32, C-14
- 'nounconnected_drive D-10
- ntb B-13
- +ntb_cache_dir C-3
- ntb_cmp B-14
- +ntb_debug_on_error C-3
- ntb_define B-13
- +ntb_enable_solver_trace_on_failure C-3
- +ntb_enable_solver_trace C-3
- +ntb_exit_on_error C-4
- ntb_fileext B-13
- ntb_incdir B-14
- +ntb_load C-4
- ntb_noshell B-14
- ntb_opts B-14
 - print_deps B-15
 - rvm B-15
- ntb_opts no_file_by_file_pp 13-29
- +ntb_random_seed C-4
- +ntb_random_seed_automatic C-4
- ntb_sfname B-16
- ntb_shell_only B-16
- ntb_sname B-16
- +ntb_solver_mode C-5
- ntb_spath B-17
- +ntb_stop_on_error C-5
- ntb_vipext 13-29, B-17
- ntb_vl B-17
- +NTC2 11-50, B-35

O

- o name 2-9, B-62
- O number B-49
- Oo B-48
- +old_ntc B-35
- operating system commands, executing D-27
- +optconfigfile 10-6, B-19
- output

C/C++ function argument direction 23-43
+overlap 11-54, B-35
+override_model_delays C-25
-override_timescale B-54

P

-P pli.tab 23-19, B-38
parallel compilation B-8, B-48
-parameters 2-8, 4-6, B-52
+pathpulse B-32
PATHPULSE\$ specparam, enabling B-32
\$period D-35
placement element
 after 15-12
 around 15-12
-platform B-62
PLI specifications
 args 23-8
 call 23-7
 check 23-7
 data 23-7
 maxargs 23-8
 minargs 23-8
 misc 23-7
 nocelldefinepli 23-8
 size 23-8
PLI table file 23-6–23-20
+pli_unprotected B-50
pli.tab file 23-6–23-20
+plusarg_ignore B-37
+plusarg_save B-36
plusargs, checking for on the simv command
line D-40
+plus-options C-25
pointer
 C/C++ function argument type 23-44
 C/C++ function return type 23-43
 input argument type 23-56
 output and inout argument type 23-56
port coercion 3-16
Port Mapping File
 VCS/SystemC cosimulation interface 22-37

‘portcoerce D-7
post-processing, options 8-3
-PP D-19
print_deps argument to -ntb_opts B-15
\$printtimescale D-33
priority keyword 14-18
procedure_prototype
 example 15-29, 15-30
+prof B-35
+protect file_suffix B-50
‘protect D-7
‘protected D-7
prx ACC capability 23-12
+pulse_e/number 11-10, 11-12, 11-14, 11-19,
11-20, B-33
+pulse_int_e 11-9, 11-10, 11-12, 11-14, B-34
+pulse_int_r 11-9, 11-10, 11-12, 11-14, B-33
+pulse_on_detect 11-20, B-34
+pulse_on_event 11-19, B-34
+pulse_r/number 11-10, 11-12, 11-14, 11-19,
11-20, B-33
pulses
 filtering out narrow pulses B-33
 and flag as error B-33
+putprotect+target_dir B-50
-pvalue 2-7, 4-5, B-52

Q

-q 2-8, B-42, C-15
\$q_add D-37
\$q_exam D-38
\$q_full D-38
\$q_initialize D-38
\$q_remove D-38

R

-R 2-7, B-19
r ACC capability 23-12, 23-17
race conditions
 avoiding 3-2–3-8

continuous assignment evaluations 3-6
 in counting events 3-7
 in flip-flops 3-5
 setting a value twice at the same time 3-4
 time zero 3-8
 using and setting a value at the same time 3-3
+rad 10-6, B-18
radix
 binary 8-25
 hexadecimal 8-25
radix, user-defined 8-29
\$random 3-37, D-39
\$readmemb D-32
\$readmemh D-32
real
 C/C++ function argument type 23-43
 input argument type 23-56
 output and inout argument type 23-56
\$realtime D-38
\$realtobits D-28
\$recovery D-35
\$recrém D-36
reg
 C/C++ function argument type 23-43
 C/C++ function return type 23-42
 input argument type 23-56
 output and inout argument type 23-56
\$reset D-39
\$reset_count D-40
\$reset_value D-40
'resetall D-3
resetting
 keeping track of the number of resets D-40
 passing a value from before to after a reset D-40
 resetting VCS to simulation time 0 D-39
\$restart D-42
 return range of a C/C++ function 23-41
 return type of a C/C++ function 23-41, 23-42
 RTL Verilog example 13-7
\$rtoi D-29
rvm B-15

rw ACC capability 23-12, 23-17

S

s ACC capability 23-12
\$save D-41
scalar
 direct access for C/C++ functions
 formal parameter type 23-54
scalar signals 8-25
scalar*
 direct access for C/C++ functions
 formal parameter type 23-54, 23-55
schematic views 8-38
Scope Types
 example 8-12
Scopes
 example 8-12
script, running 8-3
\$sdf_annotation D-40
+sdf_nocheck_celltype B-31
+sdfprotect file_suffix B-50
+sdfverbose C-16
 searching in the Waveform pane 8-34
selecting
 scopes 8-13
sequential devices
 inferring 3-19–3-23, 11-25–11-30
\$setup D-36
\$setuphold D-37
 signal dividers, adding 8-27
signal groups 8-15
signals
 scalar 8-25
 vector 8-25
simulation state
 saving D-41
size PLI specification 23-8
\$skew D-37
Smart Order 28-1
SOMA 30-2
sparse memory models 3-28

specify blocks
 disabling for an instance 11-25
 suppressing B-32
 in specific module instances 11-25
\$readmemb D-32
\$readmemh D-32
 starting DVE 8-9
\$stimen D-38
\$stop D-33
string
 C/C++ function argument type 23-44
 C/C++ function return type 23-43
 input argument type 23-56
 output and inout argument type 23-56
\$strobe D-29
sub-expressions
 in condition coverage 12-4
-sverilog B-9
SWIFT SmartModels
 generating a template B-41
\$sync\$nor\$plane D-37
-sysc B-58
 syscan utility 22-8–22-11
\$system D-27
system tasks D-10–D-43
 IEEE standard system tasks not implemented D-43
SystemC
 cosimulating with Verilog 1-2, 22-1
\$systemf D-27
 SystemVerilog assertions 20-1–??
+systemverilogext B-17

T
-t 13-5
 tb_timescale argument to -ntb_opts B-15, B-16
tchk ACC capability 23-13
\$test\$plusargs D-40
 testbench template 13-7
\$time D-38
\$timeformat D-33

-timescale B-53
'timescale D-8
timing check system tasks
 disabling
 in specific module instances 11-25
timing check system tasks, disabling B-32
timing checks
 disabling for an instance 11-25
Timopt
 the timing optimizer 11-25–11-30
+timopt 11-26
Tips button 8-34
TMPDIR A-3
toggle coverage
 defined 12-6
tokens.v file 20-16, B-50
top-level Verilog Module 13-7
+transport_int_delays 11-9, 11-12, 11-14, B-31
+transport_path_delays 11-9, 11-12, 11-14, B-31
+typdelays B-30, C-22

U

U
 direct access for C/C++ functions
 formal parameter type 23-55
-u B-61
U*
 direct access for C/C++ functions
 formal parameter type 23-54
UB*
 direct access for C/C++ functions
 formal parameter type 23-54, 23-55
-ucli 2-13, B-28, C-17
unaccelerated
 definitions and declarations 3-18–3-19
 structural instance declarations 3-18
'unconnected_drive D-10
'undef D-5
\$ungetc D-32

uniq_prior_final compiler switch 14-18	vc_Index2() 23-115
unique keyword 14-18	vc_Index3() 23-115
uniquifying identifier codes in VCD files 7-26	vc_is2state() 23-72
upper case characters, changing all identifiers to B-61	vc_is2stVector() 23-74
upper timescale 8-30	vc_is4state() 23-70
use_sigprop B-16	vc_is4stVector() 23-73
use_sigprop argument to -ntb_opts B-16	vc_isMemory() 23-69
-use_vpiobj 23-30, B-38	vc_isScalar() 23-67
'uselib D-9	vc_isVector() 23-68, 23-116
utility, vcsplit 7-50	vc_mdaSize() 23-115
 	vc_MemoryElemRef() 23-95
V	vc_MemoryRef() 23-93
-V 2-8, B-42, C-16	vc_MemoryString() 23-107
-v 2-3, B-4	vc_MemoryStringF() 23-109
+v2k B-58	vc_put2stMemoryVector() 23-105
Value column (Signal pane) 8-25	vc_put2stVector() 23-92
value transitions 8-30	vc_put4stMemoryVector() 23-104
\$value\$plusargs 5-5	vc_put4stVector() 23-90
+vc 23-120, B-39	vc_putInteger() 23-86
vc_2stVectorRef() 23-88	vc_putMemoryInteger() 23-102
vc_4stVectorRef() 23-86	vc_putMemoryScalar() 23-100
vc_argInfo() 23-113	vc_putMemoryValue() 23-106
vc_arraySize() 23-75	vc_putMemoryValueF() 23-106
vc_FillWithScalar() 23-110	vc_putPointer() 23-83
vc_get2stMemoryVector() 23-105	vc_putReal() 23-80
vc_get2stVector() 23-92	vc_putScalar() 23-76
vc_get4stMemoryVector() 23-102	vc_putValue() 23-81
vc_get4stVector() 23-90	vc_putValueF() 23-82
vc_getInteger() 23-86	vc_StringToVector() 23-85
vc_getMemoryInteger() 23-100	vc_toChar() 23-76
vc_getMemoryScalar() 23-99	vc_toInteger() 23-76
vc_getPointer() 23-83	vc_toString() 23-78
vc_getReal() 23-81	vc_toStringF() 23-79
vc_getScalar() 23-75	vc_VectorToString() 23-86
vc_handle	vc_width() 23-75
definition 23-64	vcat utility 7-36
using 23-64–23-66	-vcd filename C-19
vc_hdrs.h file 23-62	VCD+ 7-2
vc_Index() 23-114	Advantages 7-2
	System Tasks
	\$vcplusdeltacycleoff 7-21

\$vcdplusdeltacycleon 7-20
\$vcdplusmemoff 7-9
\$vcdplusmemon 7-9
\$vcdplusmemorydump 7-9
+vcdfile B-28
vcdiff utility 7-27
 syntax 7-28
vcddpost utility 7-24
 syntax 7-26
VCS 4-11
 predefined text macro D-4
VCS MX V2K Configurations and Libmaps
4-11
+vcs+boundscheck B-59
+vcs+dumpoff+t+ht C-20
+vcs+dumpon+t+ht C-20
+vcs+finish 5-24, C-15
+vcs+flush+all C-23
+vcs+flush+dump C-23
+vcs+flush+fopen C-23
+vcs+flush+log C-23
+vcs+ignorestop C-25
+vcs+initmem B-18
+vcs+initreg B-18
+vcs+learn+pli 23-24–23-28, C-24
+vcs+lic+vcsi C-23
+vcs+lic+wait 2-3, B-45, C-24
+vcs+mipd+noalias C-26
+vcs+nostdout C-16
+vcs+saif_libcell 24-2
+vcs+stop 5-24, C-15
VCS_CC A-4
VCS_COM A-4
VCS_LIC_EXPIRE_WARNING A-4
VCS_LOG A-4
'vcs_mipdexpanD D-6
VCS_NO_RT_STACK_TRACE A-4
VCS_SWIFT_NOTES A-5
+vcsi+lic+vcs C-23
+vcsi+lic+wait B-45
vcsplit utility 7-50
vec32
 storing four state Verilog data 23-49
vec32*
 direct access for C/C++ functions
 formal parameter type 23-54, 23-55
vector signals 8-25
vera_portname argument to -ntb_opts B-16
Verilog model, example 13-7
Verilog module 13-7
Verilog module description 13-15
+verilog1995ext B-18
+verilog2001ext B-18
version 8-2
version, check for 8-2
violation windows
 using multiple non-overlapping 11-54–11-59
vlogan utility 22-20–22-21
void
 C/C++ function return type 23-43
void*
 direct access for C/C++ functions
 formal parameter type 23-54
void**
 direct access for C/C++ functions
 formal parameter type 23-54
VPD
 Command line options
 Ignore \$vcdplus calls in code C-19
VPD files D-18
+vpdfiBle B-28
+vpdfiBleswitchsize B-28
+vpi B-38
-Vt B-42

W

+warn 3-38, B-43
\$warning D-11
Waveform pane
 cursors 8-31
\$width D-37
wn ACC capability 23-12

\$write D-29
\$writememb D-33
\$writememh D-33

-Xmangle B-50
-Xnoman B-51
-Xnomangle B-51

X

-xlrm uniq_prior_final compile switch 14-18
-Xman B-50

Y

-y 2-4, B-4