

VCS[®]/VCSi[™] User Guide

Version Y-2006.06-SP2
March 2008

Comments?
E-mail your comments about Synopsys
documentation to vcs_support@synopsys.com

SYNOPTSYS[®]

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, Arcadia, C Level Design, C2HDL, C2V, C2VHDL, Cadabra, Calaveras Algorithm, CATS, CRITIC, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSiM, HSPICE, Hypermodel, iN-Phase, in-Sync, Leda, MAST, Meta, Meta-Software, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PowerMill, PrimeTime, RailMill, RapidScript, Saber, SiVL, SNUG, SolvNet, Superlog, System Compiler, TetraMAX, TimeMill, TMA, VCS, Vera, and Virtual Stepper are registered trademarks of Synopsys, Inc.

Trademarks (™)

Active Parasitics, AFGen, Apollo, Apollo II, Apollo-DPII, Apollo-GA, ApolloGAI, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanTestchip, AvanWaves, BCView, Behavioral Compiler, BOA, BRT, Cedar, ChipPlanner, Circuit Analysis, Columbia, Columbia-CE, Comet 3D, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, Cyclelink, Davinci, DC Expert, DC Professional, DC Ultra, DC Ultra Plus, Design Advisor, Design Analyzer, Design Vision, DesignerHDL, DesignTime, DFM-Workbench, Direct RTL, Direct Silicon Access, Discovery, DW8051, DWPCI, Dynamic-Macromodeling, Dynamic Model Switcher, ECL Compiler, ECO Compiler, EDAnavigator, Encore, Encore PQ, Evaccess, ExpressModel, Floorplan Manager, Formal Model Checker, FoundryModel, FPGA Compiler II, FPGA Express, Frame Compiler, Galaxy, Gatran, HANEX, HDL Advisor, HDL Compiler, Hercules, Hercules-Explorer, Hercules-II, Hierarchical Optimization Technology, High Performance Option, HotPlace, HSiM^{plus}, HSPICE-Link, iN-Tandem, Integrator, Interactive Waveform Viewer, i-Virtual Stepper, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, JVXtreme, Liberty, Libra-Passport, Library Compiler, Libra-Visa, Magellan, Mars, Mars-Rail, Mars-Xtalk, Medici, Metacapture, Metacircuit, Metamanager, Metamixsim, Milkyway, ModelSource, Module Compiler, MS-3200, MS-3400, Nova Product Family, Nova-ExploreRTL, Nova-Trans, Nova-VeriLint, Nova-VHDLint, Optimum Silicon, Orion_ec, Parasitic View, Passport, Planet, Planet-PL, Planet-RTL, Polaris, Polaris-CBS, Polaris-MT, Power Compiler, PowerCODE, PowerGate, ProFPGA, ProGen, Prospector, Protocol Compiler, PSMGen, Raphael, Raphael-NES, RoadRunner, RTL Analyzer, Saturn, ScanBand, Schematic Compiler, Scirocco, Scirocco-i, Shadow Debugger, Silicon Blueprint, Silicon Early Access, SinglePass-SoC, Smart Extraction, SmartLicense, SmartModel Library, Software, Source-Level Design, Star, Star-DC, Star-MS, Star-MTB, Star-Power, Star-Rail, Star-RC, Star-RCXT, Star-Sim, Star-SimXT, Star-Time, Star-XP, SWIFT, Taurus, TimeSlice, TimeTracker, Timing Annotator, TopoPlace, TopoRoute, Trace-On-Demand, True-Hspice, TSUPREM-4, TymeWare, VCS Express, VCSi, Venus, Verification Portal, VFormal, VHDL Compiler, VHDL System Simulator, 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.
All other product or company names may be trademarks of their respective owners.

Contents

1. Getting Started	
What VCS Supports	1-3
Main Components of VCS	1-3
VCSi	1-6
Preparing to Run VCS	1-6
Obtaining a License	1-7
Setting Up Your Environment	1-8
Setting Up Your C Compiler	1-9
VCS Workflow	1-10
Compiling the Simulation Executable	1-13
Basic Compile-Time Options	1-14
Running a Simulation	1-18
Basic Runtime Options	1-19
Accessing the Discovery AMS Documentation	1-20
Making a Verilog Model Protected and Portable	1-22

2. Modeling Your Design

Avoiding Race Conditions	2-2
Using and Setting a Value at the Same Time	2-2
Setting a Value Twice at the Same Time	2-3
Flip-Flop Race Condition	2-4
Continuous Assignment Evaluation	2-5
Counting Events.	2-6
Time Zero Race Conditions	2-7
Optimizing Testbenches for Debugging.	2-8
Conditional Compilation.	2-9
Enabling Debugging Features At Runtime.	2-10
Combining the Techniques	2-13
Avoiding the Debugging Problems From Port Coercion	2-14
Creating Models That Simulate Faster	2-15
Unaccelerated Data Types, Primitives, and Statements	2-16
Inferring Faster Simulating Sequential Devices.	2-18
Modeling Faster always Blocks	2-22
Using the +v2k Compile-Time Option	2-23
Case Statement Behavior	2-24
Memory Size Limits in VCS.	2-25
Using Sparse Memory Models	2-25
Obtaining Scope Information.	2-27
Scope Format Specifications	2-27
Returning Information About the Scope.	2-30

Avoiding Circular Dependency	2-33
Designing With \$lsi_dumpports for Simulation and Test	2-34
Dealing With Unassigned Nets	2-35
Code Values at Time 0.	2-36
Cross Module Forces and No Instance Instantiation	2-36
Signal Value/Strength Codes	2-38
3. Compiling Your Design	
Using the vcs Command	3-2
Incremental Compilation	3-3
Triggering Recompilation.	3-4
Using Shared Incremental Compilation	3-5
The Direct Access Interface Directory.	3-7
Initializing Memories and Regs	3-8
Allowing Inout Port Connection Width Mismatches.	3-9
Using Lint	3-10
Changing Parameter Values From the Command Line.	3-12
Checking for X and Z Values in Conditional Expressions	3-14
Enabling the Checking.	3-15
Filtering Out False Negatives.	3-16
HSOPT Technology.	3-18
Making Accessing an Out of Range Bit an Error Condition.	3-20
Compiling Runtime Options Into the simv Executable.	3-21

Performance Considerations	3-23
Using Local Disks	3-23
Managing Temporary Disk Space on UNIX	3-24
Compile-Time Options That Impede or Accelerate VCS	3-25
Compiling for Debugging or Performance	3-27
64-32-Bit Cross-Compilation and Full 64-Bit Compilation	3-28
Identifying the Source of Memory Consumption	3-29
Minimizing Memory Consumption	3-30
Running a 64-32-Bit Cross-Compilation	3-31
Setting up the Compiler and Linker	3-32
Memory Setup	3-32
Specifying the Compiler, Linker, and -comp64 Option	3-33
Running a 64-Bit Compilation and Simulation	3-34
Using Radiant Technology	3-34
Compiling With Radiant Technology	3-35
Known Limitations	3-35
Potential Differences in Coverage Metrics	3-36
Compilation Performance With Radiant Technology	3-36
Applying Radiant Technology to Parts of the Design	3-36
The Configuration File Syntax	3-37
Configuration File Statement Examples	3-40
Library Mapping Files and Configurations	3-45
Library Mapping Files	3-45
Overriding the Search Order in the Library Mapping File	3-47
Specifying Multiple Library Mapping Files	3-47
Displaying Library Matching	3-47

Resolving 'include Compiler Directives	3-48
Configurations	3-48
Configuration Syntax	3-49
Hierarchical Configurations	3-51
The -top Compile-Time Option	3-52
Limitations of Configurations	3-53
4. Simulating Your Design	
Running and Controlling a Simulation	4-2
Invoking a Simulation at the Command Line	4-2
Invoking a Simulation From DVE	4-2
Save and Restart.	4-4
Save and Restart Example	4-4
Save and Restart File I/O.	4-6
Save and Restart With Runtime Options	4-6
Restarting at the CLI Prompt	4-8
Specifying a Very Long Time Before Stopping Simulation.	4-8
Passing Values From the Runtime Command Line.	4-10
How VCS Prevents Time 0 Race Conditions	4-11
Improving Performance	4-12
Profiling the Simulation	4-13
CPU Time Views	4-14
Memory Usage Views	4-24

5. Using the Discovery Visual Environment	
Overview of DVE Window Configuration	5-2
DVE Panes	5-4
Managing DVE Windows	5-4
Managing Target Panes	5-4
Docking and Undocking Windows and Panes	5-6
Dragging and Dropping Docked windows	5-7
Using the Menu Bar and Toolbar	5-7
Setting Display Preferences	5-10
6. VPD and EVCD File Generation	
Advantages of VPD	6-2
System Tasks and Functions	6-3
System Tasks to Generate a VPD File	6-3
System Tasks and Functions for Multi-Dimensional Arrays	6-7
Syntax for Specifying MDAs	6-7
Using \$vcdplusemon and \$vcdplusemoff	6-9
Using \$vcdplusememorydump	6-18
System Tasks for Capturing Source Statement Execution Data	6-19
Capturing Source Statement Execution	6-19
Source Statement System Tasks	6-21
System Tasks for Capturing Delta Cycle Information	6-22
System Tasks for Capturing Unique Event Information	6-23
Runtime Options	6-25
+vpdbufsize to Control RAM Buffer Size	6-25

+vpdfile to Set the Output File Name	6-26
+vpdfilesize to Control Maximum File Size	6-26
+vpdignore to Ignore \$vcdplus Calls in Code	6-27
+vpddrivers to Store Driver Information	6-27
+vpdnoports to Eliminate Storing Port Information	6-28
+vpdnocompress to Bypass Data Compression	6-28
+vpdnostrengths to Not Store Strength Information.	6-29
VPD Methodology	6-29
Advantages of Separating Simulation From Analysis	6-29
Conceptual Example of Using VPD System Tasks	6-30
VPD On/Off PLI Rules	6-32
Performance Tips.	6-33
EVCD File Generation.	6-35
Using the runtime option <code>-dump_evcd</code>	6-35
Using System Tasks.	6-36
7. VCD and VPD File Utilities	
The vcdpost Utility	7-2
Scalarizing the Vector Signals	7-2
Uniquifying the Identifier Codes.	7-3
The vcdpost Utility Syntax	7-4
The vcdiff Utility	7-5
The vcdiff Utility Syntax	7-6
The vcat Utility.	7-12
The vcat Utility Syntax	7-13

Generating Source Files From VCD Files	7-17
Writing the Configuration File.	7-18
The vcsplit Utility	7-23
The vcsplit Utility Syntax	7-23
The vcd2vpd Utility	7-26
The vcd2vpd Utility Syntax.	7-26
Options for specifying EVCD options	7-27
The vpd2vcd Utility	7-28
The vcd2vpd Utility Syntax.	7-28
The Command file Syntax	7-30
Limitations	7-33
The vpdmerge Utility	7-33
Restrictions	7-35
Limitations	7-36
Value Conflicts	7-36
8. Unified Command-Line Interface (UCLI)	
Compilation and Simulation Options for UCLI.	8-2
Using UCLI	8-3
UCLI Interactive Commands	8-4
UCLI Command-Alias File	8-9
Operating System Commands	8-9
9. Using the Old Command Line Interface (CLI)	
CLI Commands	9-2

Navigating the Design and Displaying Design Information . . .	9-2
Showing and Retrieving Simulation Information	9-4
Setting, Displaying and Deleting Breakpoints	9-7
Displaying Object Data Members	9-9
Setting and Printing Values of Variables	9-9
Traversing Call-stacks	9-9
Showing and Terminating Threads	9-10
Accessing Events.	9-11
Command Files	9-11
Key Files	9-13
Debugging a Testbench Using the CLI	9-13
Non-Graphical Debugging With the CLI.	9-14
 10. Post-Processing	
VPD	10-2
eVCD	10-3
Line Tracing.	10-3
Delta Cycle	10-3
 11. Race Detection	
The Dynamic Race Detection Tool	11-2
Enabling Race Detection	11-4
Specifying the Maximum Size of Signals in Race Conditions	11-5
The Race Detection Report	11-5
Post Processing the Report.	11-8

Debugging Simulation Mismatches	11-10
The Static Race Detection Tool	11-13
12. Delays and Timing	
Transport and Inertial Delays	12-2
Different Inertial Delay Implementations	12-4
Enabling Transport Delays.	12-7
Pulse Control.	12-7
Pulse Control with Transport Delays	12-9
Pulse Control with Inertial Delays	12-12
Specifying Pulse on Event or Pulse on Detect Behavior	12-16
Specifying the Delay Mode	12-20
13. SDF Backannotation	
Using SDF Files	13-2
Compiling the ASCII SDF File at Compile-Time	13-3
The \$sdf_annotate System Task	13-3
Limitations on Compiling the SDF File.	13-5
Precompiling an SDF File	13-7
Creating the Precompiled Version of the SDF file	13-7
Specifying an Alternative Name and Location	13-8
Reading the ASCII SDF File During Runtime	13-10
Performance Considerations	13-13
Replacing Negative Module Path Delays in SDF Files	13-13
Using the Shorter Delay in IOPATH Entries.	13-14

Disabling CELLTYPE Checking in SDF Files	13-15
The SDF Configuration File	13-16
Delay Objects and Constructs	13-17
SDF Configuration File Commands	13-18
SDF Example with Configuration File	13-25
Understanding the DEVICE Construct	13-28
Handling Backannotation to I/O Ports	13-30
Using the INTERCONNECT Construct	13-31
Multiple Backannotations to Same Delay Site	13-31
INTERCONNECT Delays	13-32
Multisource INTERCONNECT Delays	13-32
Omitting the +multisource_int_delays Option	13-34
Simultaneous Multiple Source Transitions	13-35
Single Source INTERCONNECT Delays	13-36
Min:Typ:Max Delays	13-37
Specifying Min:Typ:Max Delays at Runtime	13-38
Using the Configuration File to Disable Timing	13-39
Using the timopt Timing Optimizer	13-40
Editing the timopt.cfg File	13-43
Editing Potential Sequential Device Entries	13-43
Editing Clock Signal Entries	13-44
14. Negative Timing Checks	
The Need for Negative Value Timing Checks	14-2

Negative Timing Checks for XYZ.	14-2
The \$setuphold Timing Check Extended Syntax	14-7
Negative Timing Checks for Asynchronous Controls.	14-10
The \$recrem Timing Check Syntax	14-11
Enabling Negative Timing Checks.	14-13
Other Timing Checks Using the Delayed Signals	14-14
Checking Conditions	14-18
Toggling the Notifier Register	14-19
SDF Backannotation to Negative Timing Checks	14-19
How VCS Calculates Delays	14-20
Using Multiple Non-Overlapping Violation Windows	14-23
 15. SAIF Support	
Using SAIF Files	15-2
SAIF System Tasks	15-2
Typical Flow to Dump the Backward SAIF File using System Tasks	15-5
Criteria for Choosing Signals for SAIF Dumping	15-6
 16. SWIFT VMC Models and SmartModels	
SWIFT Environment Variables	16-2
Generating Verilog Templates	16-4
Modifying the Verilog Template File	16-5
Monitoring Signals in the Model Window	16-8

Using LMTV SmartModel Window Commands	16-10
Entering Commands Using the SWIFT Command Channel	16-13
Using the CLI to Access the Command Channel.	16-15
Loading Memories at the Start of Runtime	16-15
Compiling and Simulating a Model	16-16
Changing the Timing of a Model	16-16
17. Using the PLI	
Writing a PLI Application	17-3
Functions in a PLI Application	17-4
Header Files for PLI Applications	17-5
The PLI Table File	17-6
PLI Specifications	17-9
ACC Capabilities	17-11
Specifying ACC Capabilities for PLI Functions.	17-12
Specifying ACC Capabilities for VCS Debugging Features	17-17
Using the PLI Table File.	17-20
Enabling ACC Capabilities	17-21
Globally Enabling ACC Capabilities.	17-21
Enabling ACC Write Capabilities Using the Configuration File	17-22
Using Only the ACC Capabilities that You Need	17-25
Learning What ACC Capabilities are Used	17-25
Compiling to Enable Only the ACC Capabilities You Need	17-27
Limitations	17-28
Using VPI Routines	17-29

Support for the vpi_register_systf Routine	17-31
PLI Table File for VPI Routines	17-32
Integrating a VPI Application With VCS	17-32
Writing Your Own main() Routine	17-34
18. DirectC Interface	
Using Direct C/C++ Function Calls	18-3
How C/C++ Functions Work in a Verilog Environment	18-5
Declaring the C/C++ Function	18-6
Calling the C/C++ Function	18-12
Storing Vector Values in Machine Memory	18-14
Converting Strings	18-17
Avoiding a Naming Problem	18-19
Using Direct Access	18-20
Using the vc_hdrs.h File	18-27
Access Routines for Multi-Dimensional Arrays	18-28
UB *vc_arrayElemRef(UB*, U, ...)	18-28
U vc_getSize(UB*,U)	18-29
Using Abstract Access	18-29
Using vc_handle	18-30
Using Access Routines	18-31
int vc_isScalar(vc_handle)	18-32
int vc_isVector(vc_handle)	18-33
int vc_isMemory(vc_handle)	18-34
int vc_is4state(vc_handle)	18-35
int vc_is2state(vc_handle)	18-36

int vc_is4stVector(vc_handle)	18-37
int vc_is2stVector(vc_handle)	18-38
int vc_width(vc_handle)	18-39
int vc_arraySize(vc_handle)	18-40
scalar vc_getScalar(vc_handle)	18-40
void vc_putScalar(vc_handle, scalar)	18-40
char vc_toChar(vc_handle)	18-40
int vc_toInteger(vc_handle)	18-41
char *vc_toString(vc_handle)	18-42
char *vc_toStringF(vc_handle, char)	18-43
void vc_putReal(vc_handle, double)	18-44
double vc_getReal(vc_handle)	18-45
void vc_putValue(vc_handle, char *)	18-45
void vc_putValueF(vc_handle, char *, char)	18-46
void vc_putPointer(vc_handle, void*)	
void *vc_getPointer(vc_handle)	18-47
void vc_StringToVector(char *, vc_handle)	18-48
void vc_VectorToString(vc_handle, char *)	18-49
int vc_getInteger(vc_handle)	18-49
void vc_putInteger(vc_handle, int)	18-49
vec32 *vc_4stVectorRef(vc_handle)	18-50
U *vc_2stVectorRef(vc_handle)	18-51
void vc_get4stVector(vc_handle, vec32 *)	
void vc_put4stVector(vc_handle, vec32 *)	18-54
void vc_get2stVector(vc_handle, U *)	
void vc_put2stVector(vc_handle, U *)	18-55
UB *vc_MemoryRef(vc_handle)	18-56
UB *vc_MemoryElemRef(vc_handle, U indx)	18-58
scalar vc_getMemoryScalar(vc_handle, U indx)	18-61
void vc_putMemoryScalar(vc_handle, U indx, scalar)	18-62

int vc_getMemoryInteger(vc_handle, U indx)	18-62
void vc_putMemoryInteger(vc_handle, U indx, int)	18-64
void vc_get4stMemoryVector(vc_handle, U indx, vec32 *).	18-64
void vc_put4stMemoryVector(vc_handle, U indx, vec32 *).	18-66
void vc_get2stMemoryVector(vc_handle, U indx, U *) . . .	18-67
void vc_put2stMemoryVector(vc_handle, U indx, U *) . . .	18-67
void vc_putMemoryValue(vc_handle, U indx, char *) . . .	18-68
void vc_putMemoryValueF(vc_handle, U indx, char, char *)	18-68
char *vc_MemoryString(vc_handle, U indx)	18-69
char *vc_MemoryStringF(vc_handle, U indx, char)	18-70
void vc_FillWithScalar(vc_handle, scalar)	18-72
char *vc_argInfo(vc_handle)	18-74
int vc_Index(vc_handle, U, ...)	18-75
U vc_mdaSize(vc_handle, U)	18-76
Summary of Access Routines	18-77
Enabling C/C++ Functions	18-81
Mixing Direct And Abstract Access	18-83
Specifying the DirectC.h File	18-83
Useful Compile-Time Options	18-84
Environment Variables.	18-85
Extended BNF for External Function Declarations	18-85
 19. Using the VCS / SystemC Cosimulation Interface	
Usage Scenario Overview	19-4
Supported Port Data Types	19-5

Verilog Design Containing SystemC Leaf Modules	19-6
Input Files Required	19-7
Generating the Wrapper for SystemC Modules	19-8
Instantiating the Wrapper and Coding Style	19-11
Controlling Time Scale and Resolution in a SystemC Module Contained in a Verilog Design	19-13
Compiling a Verilog Design Containing SystemC Modules	19-14
Using GNU Compilers on Sun Solaris	19-14
Using GNU Compilers on Linux	19-15
SystemC Designs Containing Verilog Modules	19-15
Input Files Required	19-16
Generating the Wrapper	19-17
Instantiating the Wrapper	19-19
Compiling a SystemC Design Containing Verilog Modules	19-20
Elaborating the Design	19-21
Considerations for Export DPI Tasks	19-22
Use syscan -export_DPI <function-name>	19-22
Use a Stubs File	19-23
Specifying Runtime Options to the SystemC Simulation	19-24
Using GNU Compilers on SUN Solaris	19-25
Using GNU Compilers on Linux	19-25
Using a Port Mapping File	19-26
Using a Data Type Mapping File	19-27
Debugging the SystemC Portion of a Design	19-29
Debugging the Verilog Code	19-29
Debugging Both the Verilog and SystemC Portions of a Design	19-30

Transaction Level Interface	19-31
Interface Definition File	19-33
Generation of the TLI Adapters	19-36
Transaction Debug Output.	19-37
Instantiation and Binding	19-38
Supported Data Types of Formal Arguments.	19-40
Miscellaneous	19-42
Using the Built-in SystemC Simulator	19-42
Using a Customized SystemC Installation.	19-43
20. Using OpenVera Assertions	
Introducing OVA	20-2
Built-in Test Facilities and Functions	20-2
Using OVA Directives.	20-3
How Sequences Are Tested Using the assert Directive.	20-4
How Event Coverage Is Tested Using the cover Directive	20-6
OVA Flow.	20-7
Checking OVA Code With the Linter Option	20-8
Applying General Rules with VCS	20-8
Linter General Rule Messages.	20-9
Applying Magellan Rules for Formal Verification	20-16
Linter General Rule Messages:	20-16
Compiling Temporal Assertions Files	20-19
OVA Runtime Options	20-21
Functional Code Coverage Options.	20-24

OpenVera Assertions Post-Processing	20-24
OVAPP Flow	20-25
Building and Running a Post-Processor	20-26
OVA Post-Processing CLI Commands	20-31
Using Multiple Post-Processing Sessions	20-32
Multiple OVA Post-Processing Sessions in One Directory . .	20-32
Viewing Output Results	20-41
Viewing Results in a Report File	20-41
Viewing Results with Functional Coverage	20-42
Using the Default Report	20-42
Assertion and Event Summary Report	20-44
Command Line Options	20-45
Customizing the Report with Tcl Commands	20-47
Using OVA with Third Party Simulators	20-48
Inlining OVA in Verilog	20-48
Specifying Pragmas in Verilog	20-49
Methods for Inlining OVA	20-50
Unit Instantiation Using the Unit-Based Checker Library .	20-52
Instantiating Context-Independent Full Custom OVA	20-54
Template Instantiation Using the Template-Based Checker Library	20-56
Inlining Context-Dependent Full Custom OVA	20-58
Case Checking	20-59
Context-Dependent Assertion Pragmas	20-60
General Inlined OVA Coding Guidelines	20-62
Using Verilog Parameters in OVA Bind Statements	20-63

Use Model	20-63
Enabling Verilog Parameter Expansion	20-64
Limitations on the Input	20-64
Recommended Methodology	20-66
Caveats	20-66
Post-processing Flow.	20-67
Use Model	20-67
OVA System Tasks and Functions	20-68
Setting and Retrieving Category and Severity Attributes.	20-69
Starting and Stopping the Monitoring of Assertions.	20-70
Global Monitoring	20-70
Category and Severity-Based Monitoring.	20-73
Name-Based Monitoring.	20-73
Controlling the Response To an Assertion Failure.	20-74
Display Custom Message For an Assertion Failure.	20-75
Task Invocation From the CLI	20-76
Debug Control Tasks	20-77
Calls From Within Code.	20-78
Developing a User Action Function	20-82
21. OpenVera Native Testbench	
Major Features Supported in Native Testbench OpenVera.	21-3
High-level Data Types	21-3
Flow Control.	21-3
Other Features.	21-4
Getting Started With Native Testbench OpenVera	21-5
Basics of an OpenVera Testbench.	21-6

Preprocessor Directives	21-6
Top Level Constructs	21-7
Program Block	21-7
"Hello World!"	21-8
The Template Generator	21-9
Multiple Program Support	21-11
Configuration File Model	21-11
Configuration File	21-11
Use Model for Multiple Programs	21-12
Compiling Multiple Programs	21-13
NTB Options and the Configuration File	21-15
Summary	21-19
Example Configuration File	21-19
Compiling and Running the OpenVera Testbench	21-24
Compiling the Testbench with the OpenVera Design	21-24
Compiling the Testbench Separate From the OpenVera Design	21-25
Separate Compilation of Testbench Files for VCS	21-26
Compiling the Design, the Testbench Shell And the Top-level Verilog Module	21-27
Loading the Compiled Testbench On simv	21-28
Limitations	21-29
Compile-time Options	21-29
Runtime Options	21-37
Class Dependency Based OpenVera Source File Reordering	21-41
Circular Dependencies	21-43
Dependency-based Ordering in the Presence of Encryption	21-43

Using Encrypted Files	21-44
Testbench Functional Coverage	21-45
Coverage Models Using Coverage Groups	21-46
Measuring Coverage	21-49
Controlling Coverage Collection Globally	21-51
Unified Coverage Reporting.	21-53
Coverage Reporting Flow.	21-54
Persistent Storage of Coverage Data and Post-Processing Tools	21-56
Unified Coverage Directory and Database Control	21-56
Loading Coverage Data.	21-58
Solver Choice	21-61
Automatic Solver Orchestration	21-62
Temporal Assertions	21-63
Temporal Assertion Flow	21-65
Adding Assertion Objects to a Testbench.	21-65
Including the Header Files	21-66
Setting Up the AssertEngine Object.	21-66
Controlling Assertion Reporting	21-66
Resetting Assertion	21-67
Instantiating Assertion Objects.	21-67
Controlling Evaluation Attempts	21-68
Counting Successes and Failures	21-68
Setting Up the AssertEvent Objects.	21-69
Instantiating AssertEvent Objects	21-69
Suspending Threads	21-70
Eliminating AssertEvent Objects	21-70
Terminating the AssertEngine	21-71

Example Testbench	21-71
Running OpenVera Testbench with OVA	21-74
Running OpenVera Testbench with SVA	21-74
Running OpenVera Testbench with SVA and OVA Together .	21-75
OpenVera-SystemVerilog Testbench Interoperability	21-75
Scope of Interoperability	21-76
Importing OpenVera types into SystemVerilog	21-77
Data Type Mapping	21-80
Mailboxes and Semaphores	21-81
Events	21-83
Strings	21-83
Enumerated Types	21-83
Integers and Bit-Vectors	21-86
Arrays	21-87
Structs and Unions	21-88
Connecting to the Design	21-89
Mapping Modports to Virtual Ports	21-89
Semantic Issues with Samples, Drives, and Expects ...	21-93
Miscellaneous Issues	21-94
Blocking Functions in OpenVera	21-94
The terminate, wait_child, disable fork, and wait fork Constructs	21-94
Constraints and Randomization	21-94
Functional Coverage	21-95
Use Model	21-96
Using Reference Verification Methodology with OpenVera	21-98
Limitations	21-98

Testbench Optimization	21-99
NTB Performance Profiler	21-99
Enabling the NTB Profiler.	21-100
Performance Profiler Example	21-100
VCS Memory Profiler	21-105
Use Model	21-106
UCLI Interface	21-106
CLI Interface.	21-107
Incremental Profiling	21-107
Only Active Memory Reported	21-108
VCS Dynamic Memory Profile Report	21-108

22. SystemVerilog Design Constructs

SystemVerilog Data Types	22-2
Variable Data Types for Storing Integers	22-2
The chandle Data Type	22-3
User-Defined Data Types.	22-5
Enumerations.	22-5
Methods for Enumerations	22-6
The \$typeof System Function	22-8
Structures and Unions	22-10
Structure Expressions	22-13
SystemVerilog Arrays.	22-14
Multiple Dimensions	22-15
Indexing and Slicing Arrays	22-16
SystemVerilog Testbench Constructs Outside Programs	22-18
Writing To Variables.	22-19

Force and Release on SystemVerilog Variables	22-20
Automatic Variables	22-21
Multiple Drivers.	22-22
Release Behavior.	22-23
Integer Data Types	22-24
Unpacked Arrays	22-26
Structures	22-27
Using the VPI	22-28
SystemVerilog Operators	22-30
New Procedural Statements	22-31
The unique and priority Keywords in if and case Statements	22-31
The do while Statement	22-34
SystemVerilog Processes	22-35
The always_comb Block	22-35
The always_latch Block	22-38
The always_ff Block.	22-38
The final Block	22-39
Tasks and Functions	22-39
Tasks	22-40
Functions	22-41
Passing Arguments by Setting Defaults.	22-44
SystemVerilog Packages.	22-46
Exporting Time Consuming User-Defined Tasks with the SystemVerilog DPI.	22-50
Hierarchy	22-54

The \$root Top-Level Global Declaration Space	22-54
New Data Types for Ports	22-56
Instantiation Using Implicit .name Connections	22-58
Instantiation Using Implicit .* Connections	22-58
New Port Connection Rules for Variables	22-59
Ref Ports on Modules	22-60
Interfaces	22-62
Using Modports	22-66
Functions In Interfaces	22-68
Enabling SystemVerilog	22-69
Disabling unique And priority Warning Messages	22-69

23. SystemVerilog Assertion Constructs

Immediate Assertions	23-2
Concurrent Assertions Overview	23-3
Sequences	23-3
Using Formal Arguments In A Sequence	23-5
Specifying a Range of Clock Ticks	23-5
Unconditionally Extending a Sequence	23-6
Using Repetition	23-6
Specifying a Clock	23-9
Value Change Functions	23-9
Anding Sequences	23-10
Intersecting Sequences (And With Length Restriction)	23-11
Oring Sequences	23-11
Only Looking For the First Match Of a Sequence	23-12

Conditions for Sequences	23-12
Specifying That Sequence Match Within Another Sequence	23-13
Using the End Point of a Sequence	23-13
Level Sensitive Sequence Controls	23-14
Properties	23-17
Using Formal Arguments in a Property	23-18
Implications	23-19
Inverting a Property	23-21
Past Value Function	23-22
The disable iff Construct.	23-22
assert Statements	23-23
assume Statements	23-24
cover Statements	23-25
Action Blocks	23-28
Binding An SVA Module To A Design Module	23-29
Parameter Passing In A bind Directive.	23-31
The VPI For SVA	23-32
SystemVerilog Assertion Local Variable Debugging	23-33
Controlling How VCS Uses SystemVerilog Assertions	23-35
Compile-Time And Runtime Options	23-36
Ending Simulation at a Number of Assertion Failures	23-41
Disabling SystemVerilog Assertions at Compile-Time	23-42
Entering SystemVerilog Assertions as Pragmas	23-42
Options for SystemVerilog Assertion Coverage.	23-44
Reporting On Assertions Coverage	23-45

Tcl Commands For SVA And OVA Functional Coverage Reports	23-49
The assertCovReport Report Files	23-56
The report.index.html File	23-57
The tests.html File	23-62
The category.html File	23-62
The hier.html File	23-63
Assertion Monitoring System Tasks	23-64
Assertion System Functions	23-68
Using Assertion Categories	23-68
Using OpenVera Assertion System Tasks	23-69
Using Attributes	23-70
Stopping And Restarting Assertions By Category	23-71

24. SystemVerilog Testbench Constructs

Enabling Use of SystemVerilog Testbench Constructs	24-1
VCS Flow for SVTB	24-1
Options For Compiling and Simulating SystemVerilog Testbench Constructs	24-2
Compile-Time Options	24-2
Runtime Options	24-3
Compile Time or Runtime Options	24-4
The string Data Type	24-5
String Manipulation Methods	24-5
String Conversion Methods	24-8
Predefined String Methods	24-12
Program Blocks	24-15

Arrays	24-20
Dynamic Arrays	24-20
The new[] Built-In Function	24-20
The size() Method	24-22
The delete() Method	24-22
Assignments to and from Dynamic Arrays	24-22
Associative Arrays	24-24
Wildcard Indexes	24-25
String Indexes	24-25
Associative Array Assignments and Arguments	24-26
Associative Array Methods	24-26
Queues	24-29
Queue Methods	24-31
The foreach Loop	24-34
Array Aggregates (Reduction/Manipulation) Methods in Constraints	24-37
Classes	24-40
Creating an Instance (object) of a Class	24-41
Constructors	24-42
Assignment, Re-naming and Copying	24-44
Static Properties	24-45
Global Constant Class Properties	24-46
Method Declarations: Out of Class Body Declarations	24-47
Class Extensions	24-49
Subclasses and Inheritance	24-49
Abstract classes	24-50
Polymorphism	24-52

Scope Resolution Operator ::	24-54
super keyword	24-55
Casting	24-56
Chaining Constructors	24-58
Accessing Class Members	24-62
Properties	24-62
.	
Methods	24-63
“this” keyword	24-64
Class Packet Example	24-66
Unpacked Structures in Classes	24-66
Random Constraints	24-68
Random Variables	24-68
Constraint Blocks	24-69
External Declaration	24-72
Inheritance	24-72
Set Membership	24-73
Weighted Distribution	24-75
Implications	24-76
if else Constraints	24-78
Global Constraints	24-79
Default Constraints	24-80
Variable Ordering	24-87
Unidirectional Constraints	24-88
Static Constraint Blocks	24-99
Randomize Methods	24-100
randomize()	24-100
pre_randomize() and post_randomize()	24-100

Controlling Constraints	24-102
Disabling Random Variables	24-105
In-line Constraints	24-108
In-line Constraint Checker	24-109
Random Number Generation	24-111
Seeding for Randomization	24-115
randcase Statements	24-116
Random Sequence Generation	24-117
RSG Overview	24-118
Production Declaration	24-119
Production Controls	24-122
Weights for Randomization	24-122
if-else Statements	24-123
case Statements	24-125
repeat Loops	24-126
break Statement	24-126
return Statement	24-127
Aspect Oriented Extensions	24-128
Aspect-Oriented Extensions in SV	24-130
Processing of AOE as a Precompilation Expansion	24-132
Weaving advice into the target method	24-137
Pre-compilation Expansion details	24-142
Precedence	24-143
Array manipulation methods	24-165
Array ordering methods	24-165
reverse()	24-165

sort()	24-166
rsort()	24-166
Array locator methods	24-167
find()	24-167
find_index()	24-168
find_first()	24-168
find_first_index()	24-169
find_last()	24-170
find_last_index()	24-170
min()	24-171
max()	24-171
unique()	24-172
unique_index()	24-173
Array reduction methods	24-173
sum()	24-173
product()	24-174
and()	24-175
or()	24-175
xor()	24-176
Interprocess Synchronization and Communication	24-177
Semaphores	24-177
Semaphore Methods	24-179
Mailboxes	24-180
Mailbox Methods	24-182
Events	24-183
Waiting for an Event	24-183
Persistent Trigger	24-184
Merging Events	24-185

Reclaiming Named Events	24-186
Event Comparison	24-187
Clocking Blocks	24-188
Clocking Block Declaration	24-188
Input and Output Skews	24-193
Hierarchical Expressions	24-194
Signals in Multiple Clocking Blocks	24-194
Clocking Block Scope and Lifetime	24-195
Clocking Block Events	24-196
Default Clocking Blocks	24-196
Cycle Delays	24-197
Input Sampling	24-198
Synchronous Events	24-199
Synchronous Drives	24-199
Drive Value Resolution	24-200
Clocking Blocks in SystemVerilog Assertions	24-200
Sequences and Properties in Clocking Blocks	24-201
SystemVerilog Assertions Expect Statements	24-202
Virtual Interfaces	24-207
Scope of Support	24-208
Virtual Interface Modports	24-208
Driving a Net Using a Virtual Interface	24-209
Virtual Interface Modports and Clocking Blocks	24-209
Array of Virtual Interface	24-211
Clocking Block	24-212

Event Expression/Structure	24-213
Null Comparison	24-213
Not Yet Implemented	24-214
Coverage	24-214
The covergroup Construct	24-215
Defining a Coverage Point	24-217
Bins for Value Ranges	24-217
Bins for Value Transitions	24-221
Specifying Illegal Coverage Point Values	24-222
Defining Cross Coverage	24-223
Defining Cross Coverage Bins	24-224
Cumulative and Instance-based Coverage	24-226
Cumulative Coverage	24-226
Instance-based Coverage	24-227
Coverage Options	24-227
Predefined Coverage Methods	24-230
Predefined Coverage Group Functions	24-230
Unified Coverage Reporting	24-237
The Coverage Report	24-238
The ASCII Text File	24-238
The HTML File	24-240
Persistent Storage of Coverage Data and Post-Processing Tools 24-241	
Unified Coverage Directory and Database Control	24-241
Loading Coverage Data	24-243
VCS NTB (SV) Memory Profiler	24-246
Use Model	24-246

UCLI Interface	24-247
CLI Interface	24-247
Incremental Profiling	24-248
Only Active Memory Reported	24-248
VCS NTB (SV) Dynamic Memory Profile Report	24-249
The Direct Programming Interface (DPI)	24-251
Limitations	24-253
Include Files	24-253
Time Consuming Blocking Tasks	24-255

25. Source Protection

Encrypting Source Files	25-3
Encrypt Using Compiler Directives	25-3
Encrypting Specified Regions	25-4
Encrypting The Entire Source Description	25-5
Encrypting SDF Files	25-9
Specifying Encrypted Filename Extensions	25-10
Specifying Encrypted File Locations	25-10
Multiple Runs and Error Handling	25-10
Permitting CLI/PLI Access to Encrypted Modules	25-11
Simulating Encrypted Models	25-12
Using the CLI	25-12
Using System Tasks	25-13
Writing PLI Applications	25-13
Mangling Source Files	25-14
Creating A Test Case	25-23

Preventing Mangling of Top-Level Modules	25-24
Appendix A. VCS Environment Variables	
Simulation Environment Variables	A-2
Optional Environment Variables	A-3
Appendix B. Compile-Time Options	
Options for Accessing Verilog Libraries	B-4
Options for Incremental Compilation	B-6
Options for Help and Documentation	B-9
Options for SystemVerilog	B-9
Options for OpenVera Native Testbench	B-11
Options for Different Versions of Verilog	B-15
Options for Initializing Memories and Regs	B-16
Options for Using Radiant Technology	B-16
Options for 64-bit Compilation	B-16
Options for Debugging	B-17
Options for Finding Race Conditions	B-20
Options for Starting Simulation Right After Compilation	B-21
Options for Compiling OpenVera Assertions (OVA)	B-22
Options for Compiling For Simulation With Vera	B-23
Options for Compiling For Coverage Metrics	B-23
Options for Discovery Visual Environment and UCLI	B-30
Options for Converting VCD and VPD Files	B-31
Options for Specifying Delays	B-32
Options for Compiling an SDF File	B-35

Options for Profiling Your Design	B-37
Options for File Containing Source File Names and Options	B-38
Options for Compiling Runtime Options into the simv Executable	B-39
Options for Pulse Filtering	B-40
Options for PLI Applications	B-41
Options to Enable and Disable Specify Blocks and Timing Checks	B-42
Options to Enable the VCS DirectC Interface	B-43
Options for Negative Timing Checks	B-43
Options for Flushing Certain Output Text File Buffers	B-44
Options for Simulating SWIFT VMC Models and SmartModels	B-45
Options for Controlling Messages	B-45
Options for Cell Definition	B-48
Options for Licensing	B-49
Options for Controlling the Assembler	B-50
Options for Controlling the Linker	B-50
Options for Controlling the C Compiler	B-51
Options for Source Protection	B-54
Options for Mixed Analog/Digital Simulation	B-56
Options for Changing Parameter Values	B-56
Checking for X and Z Values in Conditional Expressions	B-57
Options to Specify the Time Scale	B-57
General Options	B-58
Enable Verilog 2001 Features	B-58
Enable the VCS/SystemC Cosimulation Interface	B-58
Reduce Memory Consumption	B-58

TetraMAX	B-59
Make Accessing an Undeclared Bit an Error Condition ..	B-59
Treat Output Ports As Inout Ports	B-59
Allow Inout Port Connection Width Mismatches.	B-59
Specifying a VCD File.	B-59
Memories and Multi-Dimensional Arrays (MDAs)	B-60
Specifying a Log File	B-60
Hardware Modeling	B-61
Changing Source File Identifiers to Upper Case	B-61
Defining a Text Macro.	B-61
Specifying the Name of the Executable File.	B-62
Returning The Platform Directory Name	B-62
Specifying Native Code Generation	B-62
For Long Calls	B-62

Appendix C. Simulation Options

Options for Simulating OpenVera Testbenches	C-2
Options for Simulating OpenVera Assertions.	C-4
Options for SystemVerilog Assertions	C-6
Options for a CLI Command File	C-9
Options for Specifying VERA Object Files	C-10
Options for Coverage Metrics	C-10
Options for Enabling and Disabling Specify Blocks	C-12
Options for Specifying When Simulation Stops	C-13
Options for Recording Output	C-13
Options for Controlling Messages	C-14
Options for Discovery Visual Environment and UCLI	C-15
Options for VPD Files	C-15

Options for Controlling \$gr_waves System Task Operations.	C-17
Options for VCD Files	C-18
Options for Specifying Min:Typ:Max Delays	C-19
Options for Flushing Certain Output Text File Buffers	C-20
Options for Licensing	C-21
General Options.	C-22
Viewing the Compile-Time Options Used to Create the Executable	C-22
Stopping Simulation When the Executable Starts	C-22
Recording Where ACC Capabilities are Used	C-22
Suppressing the \$stop System Task	C-23
Enabling User-Defined Plusarg Options.	C-23
Enabling Overriding the Timing of a SWIFT SmartModel.	C-23
Specifying acc_handle_simulated_net PLI Routine and MIPD Annotation	C-23

Appendix D. Compiler Directives and System Tasks

Compiler Directives	D-2
Compiler Directives for Cell Definition	D-2
Compiler Directives for Setting Defaults	D-3
Compiler Directives for Macros	D-3
Compiler Directives for Detecting Race Conditions.	D-5
Compiler Directives for Delays.	D-5
Compiler Directives for Backannotating SDF Delay Values.	D-7
Compiler Directives for Source Protection.	D-7
Compiler Directives for Controlling Port Coercion	D-8
General Compiler Directives	D-8

Compiler Directive for Including a Source File	D-8
Compiler Directive for Setting the Time Scale	D-9
Compiler Directive for Specifying a Library	D-9
Compiler Directive for Maintaining The File Name and Line Numbers	D-10
Unimplemented Compiler Directives	D-10
System Tasks and Functions.	D-10
System Tasks for SystemVerilog Assertions Severity	D-11
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-25
System Tasks for Executing Operating System Commands	D-26
System Tasks for Log Files	D-27
System Tasks for Data Type Conversions	D-28
System Tasks for Displaying Information.	D-28
System Tasks for File I/O.	D-29
System Tasks for Loading Memories.	D-31
System Tasks for Time Scale.	D-32
System Tasks for Simulation Control.	D-32
System Tasks for Timing Checks.	D-33
System Tasks for PLA Modeling	D-36
System Tasks for Stochastic Analysis	D-36
System Tasks for Simulation Time.	D-37

System Tasks for Probabilistic Distribution	D-38
System Tasks for Resetting VCS.	D-38
General System Tasks and Functions	D-39
Checks for a Plusarg	D-39
SDF Files	D-39
Counting the Drivers on a Net	D-40
Depositing Values.	D-40
Fast Processing Stimulus Patterns.	D-40
Saving and Restarting The Simulation State	D-41
Checking for X and Z Values in Conditional Expressions	D-41
IEEE Standard System Tasks Not Yet Implemented in VCS	D-42

Appendix E. PLI Access Routines

Access Routines for Reading and Writing to Memories	E-2
acc_setmem_int.	E-4
acc_getmem_int	E-5
acc_clearmem_int	E-6
Examples	E-6
acc_setmem_hexstr.	E-11
Examples	E-12
acc_getmem_hexstr	E-15
acc_setmem_bitstr.	E-16
acc_getmem_bitstr.	E-17
acc_handle_mem_by_fullname	E-18
acc_readmem	E-19
Examples	E-20
acc_getmem_range.	E-21

acc_getmem_size	E-22
acc_getmem_word_int.....	E-23
acc_getmem_word_range	E-24
Access Routines for Multidimensional Arrays	E-25
tf_mdanodeinfo and tf_imdanodeinfo.....	E-26
acc_get_mda_range	E-28
acc_get_mda_word_range()	E-29
acc_getmda_bitstr().....	E-31
acc_setmda_bitstr().....	E-32
Access Routines for Probabilistic Distribution.....	E-33
vcs_random	E-34
vcs_random_const_seed.....	E-35
vcs_random_seed	E-35
vcs_dist_uniform	E-36
vcs_dist_normal.....	E-37
vcs_dist_exponential	E-38
vcs_dist_poisson	E-39
Access Routines for Returning a String Pointer to a Parameter Value E-39	
acc_fetch_paramval_str.....	E-40
Access Routines for Extended VCD Files.....	E-40
acc_lsi_dumpports_all	E-42
acc_lsi_dumpports_call	E-43
acc_lsi_dumpports_close.....	E-45
acc_lsi_dumpports_flush	E-46

acc_lsi_dumpports_limit	E-47
acc_lsi_dumpports_misc	E-48
acc_lsi_dumpports_off	E-49
acc_lsi_dumpports_on	E-50
acc_lsi_dumpports_setformat	E-52
acc_lsi_dumpports_vhdl_enable	E-53
Access Routines for Line Callbacks	E-54
acc_mod_lcb_add	E-55
acc_mod_lcb_del	E-57
acc_mod_lcb_enabled	E-58
acc_mod_lcb_fetch	E-59
acc_mod_lcb_fetch2	E-60
acc_mod_sfi_fetch	E-62
Access Routines for Source Protection	E-64
vcsSpClose	E-68
vcsSpEncodeOff	E-68
vcsSpEncodeOn	E-69
vcsSpEncoding	E-71
vcsSpGetFilePtr	E-72
vcsSpInitialize	E-73
vcsSpOvaDecodeLine	E-74
vcsSpOvaDisable	E-75
vcsSpOvaEnable	E-76
vcsSpSetDisplayMsgFlag	E-78
vcsSpSetFilePtr	E-78
vcsSpSetLibLicenseCode	E-79

vcsSpSetPliProtectionFlag	E-80
vcsSpWriteChar	E-81
vcsSpWriteString	E-83
Access Routine for Signal in a Generate Block.	E-84
acc_object_of_type	E-84
VCS API Routines	E-85
Vcsinit()	E-85
VcsSimUntil()	E-85

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 enables you to analyze, compile, and simulate Verilog 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 supports all levels of design descriptions, but is optimized for the behavioral and register transfer levels.

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

In addition, VCS supports Synopsys DesignWare IP, the VCS Verification Library, VMC models, and the Vera testbench tool.

VCS is also integrated with other third-party tools via the programming language interface (PLI). Also, the DirectC Interface enables interaction between Verilog designs and applications written in C or C++.

VCS is integrated with many third party tools such as testbench tools, memory model generation tools, acceleration and emulation systems, and graphical user interfaces.

This chapter covers the following topics:

- [What VCS Supports](#)
- [Main Components of VCS](#)
- [VCSi](#)
- [Preparing to Run VCS](#)
- [VCS Workflow](#)
- [Compiling the Simulation Executable](#)
- [Running a Simulation](#)
- [Accessing the Discovery AMS Documentation](#)
- [Making a Verilog Model Protected and Portable](#)

What VCS Supports

VCS provides fully featured implementations of the following:

- The Verilog language as defined in the *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language* (IEEE Std 1364-1995) and the *Standard Verilog Hardware Description Language* (IEEE Std 1364-2001).
- The SystemVerilog 3.1a language (with some exceptions) as defined in *SystemVerilog 3.1a Accellera's Extensions to Verilog*.

In addition, VCS supports interfaces to a variety of other simulators and models, including (but not limited to) user PLI applications conforming to IEEE Std 1363-1995, delay calculators, SDF delay annotation, and Synopsys Logic Modeling SmartModels[®].

Main Components of VCS

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

- SystemVerilog — an extension of the Verilog language that adds new design, testbench, and assertion constructs. For details on SVA, see [Chapter 23, "SystemVerilog Assertion Constructs"](#), [Chapter 22, "SystemVerilog Design Constructs"](#), and [Chapter 24, "SystemVerilog Testbench Constructs"](#).

- OpenVera Assertions (OVA) — provides an easy and concise way to describe sequences of events, and facilities to test for their occurrence. VCS natively compiles OVA. For details on OVA, see [Chapter 20, "Using OpenVera Assertions"](#) and the *OpenVera Language Reference Manual: Assertions* volume. Many of the implemented SystemVerilog assertions constructs are functionally comparable to OpenVera assertion constructs.
- OpenVera Native Testbench — a testbench language that is a subset of the OpenVera testbench language. VCS can natively compile testbench files written in OpenVera testbench constructs into the simv executable file, along with Verilog source files and OpenVera Assertions (OVA) files. For details on OpenVera Native Testbench, see [Chapter 21, "OpenVera Native Testbench"](#).
- 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 details, see [Chapter 5, "Using the Discovery Visual Environment"](#) and the *Discovery Visual Environment User Guide*. DVE is in the process of replacing VirSim.
- 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 details, see the *VCS Coverage Metrics User Guide*.

- **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.
- **Incremental Verilog Compilation** — reduces the turnaround time from design modification by minimizing the amount of recompilation. This capability enables VCS to automatically compare the current design against the previously compiled database; it then recompiles only those portions of the design that have changed. For details, see [“Incremental Compilation” on page 3-3](#).
- **64-Bit Cross-Compilation and Full 64-Bit Compilation** — VCS offers a choice of methodologies for high-capacity compilation and simulation. Its `-comp64` option invokes a cross-compilation process that compiles a design on a 64-bit machine, which can then be simulated on a 32-bit machine. The `-full64` option both compiles and simulates a design on a 64-bit machine.
- **Mixed Signal Simulation** — Synopsys provides the *Discovery AMS: NanoSim-VCS User Guide* and the *Discovery AMS: Enhanced NanoSim-VCS User Guide* to NanoSim and VCS users who need to do mixed signal simulation. See [“Accessing the Discovery AMS Documentation” on page 1-20](#).

VCSi

VCSi is offered as an alternate version of VCS. VCS and VCSi are identical except that VCS is more highly optimized, resulting in greater speed. VCS and VCSi are guaranteed to provide the exact same simulation results. VCSi implementation requirements are summarized as follows:

1. There are separate licenses for VCSi.
2. VCSi is invoked using the `vcsi` command, instead of the `vcs` command.

Note:

Hereafter, all references to VCS in this manual pertain to VCSi as well.

The `+vcsl+lic+vcs` compile-time option enables you to run VCSi with a VCS license when all VCSi licenses are in use, and the `+vcs+lic+vcsl` compile-time option enables you to run VCS with three VCSi licenses.

Preparing to Run VCS

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

- [Obtaining a License](#)
- [Setting Up Your Environment](#)
- [Setting Up Your C Compiler](#)

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/vcs6.0/license.dat
```

Note:

- Using multiple port@host in the \$LM_LICENSE_FILE can cause previous VCS releases, which use pre FLEX-LM6.1 daemons, not to work. To work around this problem, put the old port@host before the new port@host in LM_LICENSE_FILE variable or simply point to the license file instead of using port@host, for example:

```
% setenv LM_LICENSE_FILE 7400@server:7500@server
```

Here, 7400 is the port on machine "server" where the old license daemon, viewlgrd, is running, while 7500 is the port on machine "server" where the new license daemon, snpslmd, is running.

OR

```
setenv LM_LICENSE_FILE /u/edatools/oldvcs/\
viewlgrd_license.dat:/u/edatools/vcs/\
snpslmd_license.dat
```

Setting Up Your Environment

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

- \$VCS_HOME environment variable

When you or someone at your site installed VCS, the installation created a directory called the *installation_dir* directory. Set the VCS_HOME environment variable to the path of this directory as follows:

```
% setenv VCS_HOME installation_dir
```

- PATH environment variable

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)
```

Make sure the path environment variable is set to a bin directory containing a make or gmake program.

- LM_LICENSE_FILE environment variable

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 in UNIX.

For example:

```
% setenv LM_LICENSE_FILE 7182@serveroh:/u/net/serveroo/  
eda_tools/license.dat
```

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

Setting Up Your C Compiler

On Solaris, HP, and Linux, VCS requires a C compiler to link the executable file that you simulate, and, in some cases, to compile intermediate files. If this is the case, you will need to set the path to a C compiler.

Solaris does not come bundled with a C compiler so you must purchase the C compiler for Solaris or use gcc. VCS assumes the compiler is located in its default location: /usr/ccs/bin.

HP, Linux, and IBM RS/6000 AIX platforms all come bundled with a C compiler. VCS assumes the compiler is located in its default location: /usr/bin.

You can specify a different location with the `VCS_CC` environment variable or with the `-cc` compile time option.

- v**
Displays the version number and exits.
- lib**
Displays the library mapping.
- help**
Lists the options to show_setup.

VCS Workflow

The process of using VCS to simulate a design consists of the following tasks:

- [Compiling the Simulation Executable](#)
- [Running a Simulation](#)

This approach simulates faster and uses less memory than interpretive simulators. The process of compiling an executable binary avoids the extra layers and inefficiency of an interpretive simulation environment.

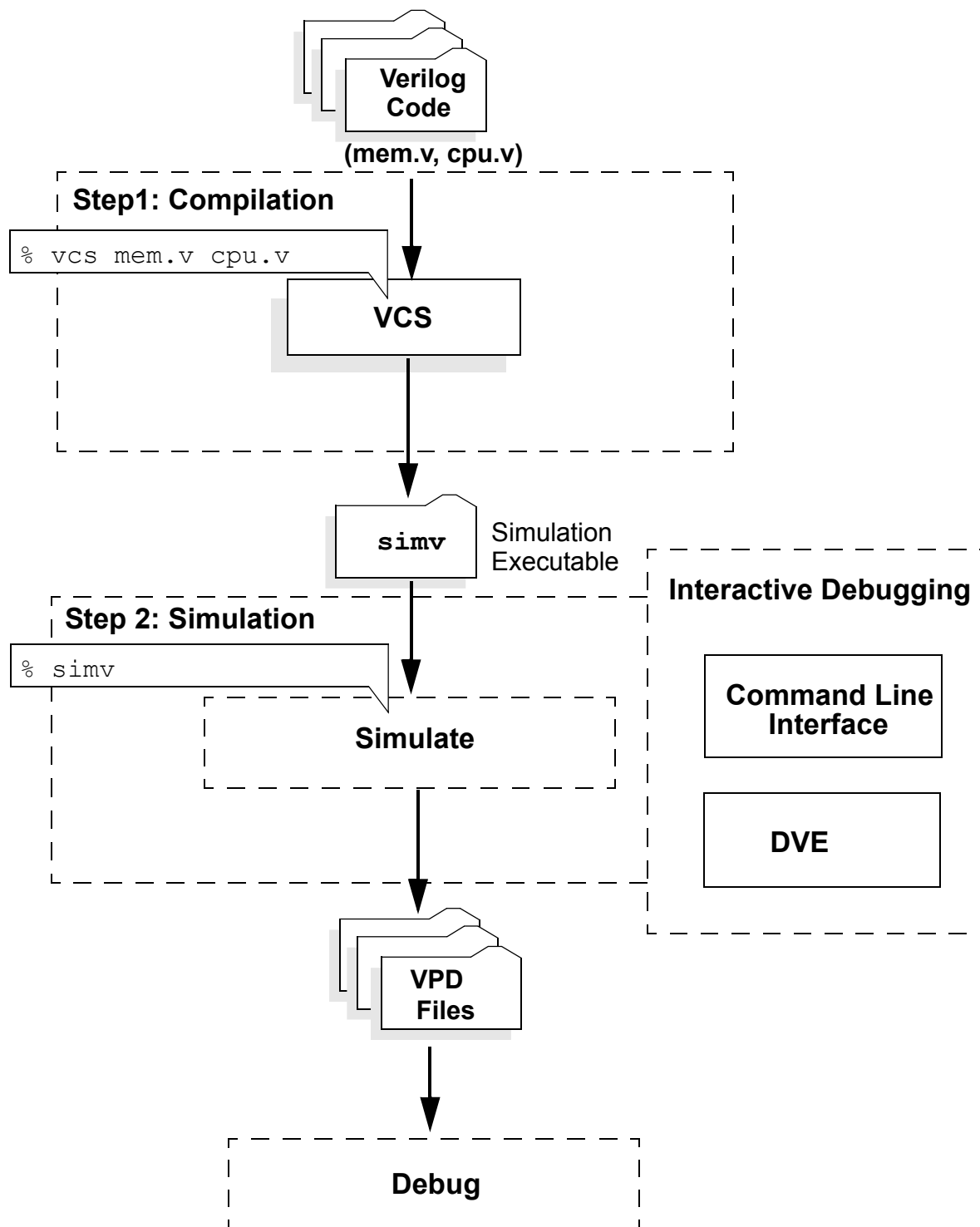
On Linux, Solaris, and HP platforms, you can use VCS to generate object code directly without generating C or assembly language files. Incremental compilation allows you to avoid compiling modules that have not changed since the last time you compiled them. For more details on incremental compilation, see [“Incremental Compilation” on page 3-3](#).

Note:

For information on coverage features, see the *VCS /VCS MX Coverage Metrics User Guide*.

Figure 1-1 illustrates the VCS workflow.

Figure 1-1 Basic VCS Compilation and Simulation Flow



Compiling the Simulation Executable

After setting up your environment and preparing your source files, you are ready to compile a simulation executable. To create this executable, named `simv` by default, use the following VCS command line:

```
vcs source_files [source_or_object_files] options
```

where:

`source_files`

The Verilog, OpenVera assertions, or OpenVera testbench source files for your design. The file names must be 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.

`options`

Compile-time options that control how VCS compiles your Verilog source files. For details, see [“Basic Compile-Time Options” on page 1-14](#).

The following is an example command line used at compile-time:

```
vcs top.v toil.v -RI +v2k
```

By default, VCS names the executable binary file `simv`. You can specify a different name with the `-o` compile-time option.

Basic Compile-Time Options

This section outlines some of the basic compile-time options you can use to control how VCS compiles your Verilog source files. Detailed descriptions and usage instructions for all compile-time options are available in [Chapter 3, "Compiling Your Design"](#) and [Appendix B, "Compile-Time Options"](#).

-cm [line|cond|fsm|tgl|path|branch]

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.

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
```

+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.

-f *filename*

Specifies a file name that contains a list of absolute pathnames for Verilog source files and compile-time options.

+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, separated by +.

-I

Compiles for interactive use and instructs VCS to automatically include `+cli` (command line interface), `-P virsims.tab` (default VirSim PLI table), and `-lm` (math library). This option enables the use of system tasks for writing VCD+ files for post-processing in VirSim.

-line

Enables source-level debugging tasks such as stepping through the code, displaying the order in which VCS executed lines in your code, and displaying the last statement executed before simulation stopped.

-l *filename*

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

+nospecify

Suppresses module path delays and timing checks in specify blocks. This option can significantly improve simulation performance.

+notimingcheck

Suppresses timing check system tasks during compilation. This option can moderately improve simulation performance. The extent of this improvement depends on the number of timing checks that VCS ignores.

-ntb

Enables the use of the OpenVera Testbench language constructs described in the *OpenVera Language Reference Manual: Native Testbench*.

-ova_file *filename*

Identifies an OVA file as input. This option is not required if the OVA file name contains a .ova extension.

-P *pli.tab*

Compiles a user-defined PLI definition table file.

-PP

Compiles a VCD file for interactive debugging while minimizing the amount of net data for fast post-processing.

-R

Runs the executable file immediately after VCS links it together. You can add any runtime option to the `vcs` command line.

-RI

Compiles model for interactive use, invokes the VirSim graphical user interface immediately after compilation, and pauses simulation at time zero.

-s

Specifies stopping simulation, and entering the CLI interactive mode, just as simulation begins. Use this option on the `vcs` command line along with the `-R` and `+cli` options. The `-s` option is also a runtime option on the `simv` command line.

-sverilog

Enables the use of SystemVerilog code.

+v2k

Enables language features in the IEEE 1364-2001 standard.

-v *filename*

Specifies a Verilog library file, in which VCS looks for the modules and UDP instances that are instantiated, but not defined, in the source code.

+vc [+abstract] [+allhdrs] [+list]

Enables the direct call of C/C++ functions in your source code using the DirectC interface. The optional suffixes specify the following:

+abstract

Specifies that you are using abstract access through `vc_handles` to the data structures for the Verilog arguments.

+allhdrs

Writes the `vc_hdrs.h` file that contains external function declarations that you can use in your Verilog code.

+list

Displays all the functions that you called in your Verilog source code.

-vera

Specifies the standard VERA PLI table file and object library.

-y *directory*

Specifies a Verilog library directory. VCS looks in the source files in this directory for the modules and UDP instances that are instantiated, but not defined, in the source code.

On Solaris, HP, and Linux machines, VCS can generate object files from your Verilog source files and does so by default. This is sometimes called native code generation. On these machines, if you enter the `-gen_asm` or `-gen_c` compile-time options, VCS generates corresponding intermediate assembly or C files and then assembles or compiles these intermediate files into object files.

On DEC Alpha, and IBM RS/6000 AIX, VCS always generates intermediate C files. The `-gen_c` compile-time option is a default option on these platforms.

Running a Simulation

To run a simulation, you simply specify the name of the executable file (produced from the compilation process) at the command line.

The command line syntax for running a simulation is as follows:

executable_file options

Here:

executable_file

The executable file that is created by the `vcs` command, which compiles your source code and links your design with VCS to form the executable.

options

Runtime options that specify how to simulate your design. Some of the basic runtime options are described in [“Basic Runtime Options” on page 1-19](#).

For example, the following command line can be used at runtime:


```
% simv -l log +notimingcheck
```

Basic Runtime Options

This section outlines some of the basic runtime options you can use to control how VCS compiles your Verilog source files. Detailed descriptions and usage instructions for all runtime options are available in [Chapter 4, "Simulating Your Design"](#) and [Appendix C, "Simulation Options"](#).

Here:

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

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

line

Monitor for line or statement coverage.

cond

Monitor for condition coverage.

fsm

Monitor for FSM coverage.

tgl

Monitor for toggle coverage.

path

Monitor for path coverage.

branch

Monitor for branch coverage

-l filename

All output of simulation is written to the file you specify as *filename*, as well as to the standard output.

+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.

-ova_cov

Enables functional coverage reporting.

-ova_report

Generates an OVA report file in addition to printing results on screen. The default file name and location is `simv.vdb/report/ova.report` but you can specify a different name and location as an argument to this option.

+vcs+learn+pli

Keeps track of where you use ACC capabilities for debugging operations so that you can recompile your design and, in the next simulation, enable them only where you need them. This is useful because ACC capabilities have a performance cost so you only want to enable them where you need them. With this option VCS writes the `pli_learn.tab` secondary PLI table file. You provide this file name with the `+applylearn` compile-time option when you recompile your design.

Accessing the Discovery AMS Documentation

For detailed information on mixed signal simulation with VCS and NanoSim, see the Discovery AMS documentation

There are two ways to access the Discovery AMS documentation:


- Using Synopsys Documentation on the Web

- Using the PDF files in the NanoSim installation

To access the Discovery AMS documentation in Synopsys Documentation on the Web:

1. Go to www.synopsys.com.
2. Click on SOLVNET.
3. Click on Documentation on the Web.
4. Click the Go button next to Browse.
5. Click on NanoSim.
6. Click on the document titles:
 - *Discovery AMS NanoSim-VCS User Guide*, or
 - *Discovery AMS Enhanced NanoSim-VCS User Guide*

Clicking on the user guide titles displays the user guide in HTML format.

Clicking on the icon for a PDF file downloads  the PDF file for this user guide.

To access the PDF files in the NanoSim installation:

1. Change directories to the *NanoSim_installation_directory/doc/ns/manuals* directory.
2. Load either of the following files into the Acrobat reader:
 - *e_ns_vcs.pdf* for the Discovery AMS: Enhanced NanoSim-VCS User Guide.
 - *ns_vcs_mx.pdf* for the Discovery AMS: NanoSim-VCS User Guide

Making a Verilog Model Protected and Portable

After you have successfully verified your design using VCS, you can use the Verilog Model Compiler (VMC) to make the design portable and protected. VMC enables you to secure your design and distribute it to your partners and internal or external customers without a Non Disclosure Agreement (NDA).

VMC is a model development tool used to generate portable models, starting with Verilog source and producing compiled SWIFT models. SWIFT is a language- and simulator-independent interface that allows your model to run with any SWIFT-compatible simulators; more than thirty simulators are now available.

VMC models contain no Verilog source code, so they protect the intellectual property of the underlying design. This enables model developers to distribute their models without revealing the contents, because the models are secure. More importantly, the models are functionally exact because they are derived from the original Verilog description of the model.

2

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 you code designs that simulate most efficiently with VCS.

This chapter covers 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. For details, see [Chapter 11, "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
endmodule
```

```

    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
    $vcdpluson(0, design_1);
    $vcdplustraceon(design_1);
    $vcdplusdeltacycleon;
    $vcdplusglitchon;
  \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
$vcpluson(0, design_1);
$vcplusraceon(design_1);
$vcplusdeltacyclone;
$vcplusglitchon;
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. So for 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. On the other hand, 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 comppostprocess
initial
  if ($test$plusargs("runpostprocess"))
    begin
      $vcdpluson(0,design_1);
      $vcdplustraceon(design_1);
      $vcsplusdeltacycleon;
      $vcdplusglitchon;
    end
  endif
`endif
```

Here both the `+define+comppostprocess` 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_indentifier(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, pull0) 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;
```

Here 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</code> and <code>`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</code> <code>\$fsanf</code> <code>\$scanf</code> and more	no
passing values from the runtime command line: <code>\$value\$plusarg</code> system function	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: <code>r1=r2**r3;</code>	yes
attributes: <code>(* optimize_power=1 *)</code> <code>module dev (res,out,clk,data1,data2);</code> generate statements	yes
localparam declarations	yes
Automatic tasks and functions <code>task automatic t1();</code>	requires the -sverilog compile-time option
constant functions <code>localparam lp1 = const_func(p1);</code>	yes
parameters with a bit range <code>parameter bit [7:0][31:0] P =</code> <code>{32'd1, 32'd2, 32'd3, 32'd4, 32'd5, 32'd6, 32'd7, 32'd8};</code>	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 ?.

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 used 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. Don't 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(inport);
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, or a named block, or a user-defined task, or a function in a module instance.

When VCS executes these system functions it does 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

```

declaration of C function named showInst

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

```

module child;
initial discriminator.t;
endmodule

```

```

module top;
child c1();
child c2();
child c3();
child c4();
endmodule

```

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

Note:

See [“Signal Value/Strength Codes” on page 2-38](#). 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)

3

Compiling Your Design

VCS compiles your design before simulation. You can use the `vcs` command to compile your designs. The resulting executable is called `simv`.

There are several options and techniques that you can use to compile your design for high performance or easy debugging. Native code generation is default in VCS.

This chapter covers the following topics:

- [Using the vcs Command](#)
- [Incremental Compilation](#)
- [Triggering Recompilation](#)
- [Using Shared Incremental Compilation](#)
- [The Direct Access Interface Directory](#)

- [Initializing Memories and Regs](#)
- [Initializing Memories and Regs](#)
- [Allowing Inout Port Connection Width Mismatches](#)
- [Using Lint](#)
- [Changing Parameter Values From the Command Line](#)
- [Checking for X and Z Values in Conditional Expressions](#)
- [Making Accessing an Out of Range Bit an Error Condition](#)
- [Compiling Runtime Options Into the simv Executable](#)
- [Performance Considerations](#)
- [64-32-Bit Cross-Compilation and Full 64-Bit Compilation](#)
- [Using Radiant Technology](#)
- [Library Mapping Files and Configurations](#)

Using the vcs Command

To compile your design, use the `vcs` command. The syntax is as follows:

```
% vcs source_files [source_or_object_files] [options]
```

For example, to compile the `top.v` and `toil.v` source files, enter the following command:

```
% vcs top.v toil.v
```

For a complete description of the syntax of the `vcs` command, see [Appendix B, "Compile-Time Options"](#).

Incremental Compilation

VCS compiles your source code on a module-by-module basis. By default, when you recompile the design, VCS recompiles only the modules that have changed since the last compilation. This is called incremental compilation.

During compilation VCS creates a subdirectory named `csrc` to store the files generated by compilation. These files are as follows:

- A makefile that controls the compilation process.
- The object files output from compilation. VCS links these files with the simulation engine to create the `simv` executable. If you use native code generation (available only on Solaris, HP, and Linux), VCS generates these files directly.
- Intermediate C or assembly files. If you are not using native code generation, VCS by default compiles or assembles the object files for the modules in your design.

In incremental compilation, when you enter a `vcs` command line, VCS compares the modules in your source files to the descriptor information in the generated files from the previous compilation. If a module's contents are different from what VCS recorded in the descriptor information, VCS recompiles the module. If the module's contents match what is recorded in the descriptor information, VCS does not recompile the module.

Compile-time options that affect incremental compilation all begin with `-M`. For more details on these options, see [“Options for Incremental Compilation” on page B-6](#).

Triggering Recompilation

VCS recompiles a module when you change its contents. The following conditions also trigger the recompilation of a module:

- Change in the VCS version.
- Changes in the command-line options.
- Change in the target of a hierarchical reference.
- Change in the ports of a module instantiated in the module.
- Change in the calls from a `$dumpvars` system task to the module.
- 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 another module in the same source file.
- Nonfunctional changes such as comments or white space.

Using Shared Incremental Compilation

Shared incremental compilation allows a team of designers working on a large design to share the generated files for a design in a central location so that they do not have to recompile the parts of a design that have been debugged and tested by other members of the team.

To invoke shared incremental compilation, your team can use the following compile-time options:

`-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.

`-Mdir=dir`

Specifies the pathname of a central directory where you want VCS to write the generated files such as option files. This option allows you to provide other members of your team with the generated files for new or improved modules in a large design so that other members do not have to compile those modules.

Suppose, for example, a board designer and a chip designer are working together on a project. The board designer will use the chip designer's chip design in the board design.

The chip designer is responsible for debugging the chip design and creating a shared design that the board designer can use to debug the board design without having to recompile the chip design.

The chip designer needs to know what debug features the board designer needs in the chip design. The chip designer meets the board designer and learns that the board designer needs to dump the signals in a memory submodule in the chip design, so the chip designer adds a `$dumpvars` system task to the chip design:

```
$dumpvars(0,asic.memory);
```

The chip designer now compiles the chip design using the following `vcs` command line:

```
% vcs -Mdir=path_to_shared_directory other_options  
source_files_for_the_chip_design
```

The chip designer includes the `-Mdir` option specifying a directory pathname into which VCS writes the generated files and makes sure that the board designer can access this directory.

The board designer first copies the source files for the chip design from the chip designer and then compiles the board design with the following command line:

```
% vcs -Mdir=local_dir -Mlib=path_to_shared_directory  
other_options source_files_for_the_chip_design  
source_files_for_the_board_design
```

VCS does not recompile the chip design because the board designer included the `-Mlib` option telling VCS where to look for descriptor information to compare to the chip design source files. This descriptor information tells VCS that it doesn't need to recompile the chip design so VCS uses the generated files in this directory to create the `simv` executable for simulating the board design

The board designer includes the `-Mdir` option to specify a directory where VCS writes the generated files for the board design but not the chip design. VCS also uses the generated files in this directory to create the `simv` executable for simulating the board design. If the board designer omits this option, VCS will write the generated files to a `csrc` directory in the board designer's current directory.

For shared incremental compilation to be possible, the board designer and the chip designer must use the same version of VCS. If they use different versions, VCS recompiles the chip design.

The Direct Access Interface Directory

When you compile your design with certain options, VCS creates a `simv.daidir` in the same location as the `simv` executable file.

This directory contains database files that describe the structure and content of your design. VCS uses this database for debugging tasks such as stepping through the code and the depositing or forcing values. VCS also uses this database for PLI ACC capabilities.

By default this directory is named `simv.daidir` and VCS creates it in the current directory.

If you include the `-o` compile-time option to change the location and name of the executable file, VCS creates the Direct Access Interface directory in the same directory as that executable file. It also gives that Direct Access Interface directory the same name as the executable file but with the `.daidir` file name extension.

Initializing Memories and Regs

VCS has compile-time options for initializing all bits of 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 Verilog portion of the design.

The `+vcs+initmem` option initializes regular memories and multi-dimensional 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;
:
:
```

Allowing Inout Port Connection Width Mismatches

By default it is an error condition if you connect a signal to an inout port and that signal does not have the same bit width as the inout port. It is a warning condition if you connect such a mismatched signal to an input or output port.

You can use the `+noerrorIOPCWM` compile-time option to change the error condition for an inout port to a warning condition, and thereby allow VCS to create the `simv` executable. Consider the following code:

```
module test;
wire [7:0] w1;
:
dev dev1 (w1);
:
endmodule

module dev(gk);
inout [15:0] gk;
:
endmodule
```

Without the `+noerrorIOPCWM` compile-time option, VCS displays the following error message and does not create the `simv` executable:

```
Error-[IOPCWM] Inout Port connection width mismatch
    The following 8-bit expression is connected to 16-bit
port "gk" of
    module "dev", instance "dev1".
```

If you include the `+noerrorIOPCWM` compile-time option, VCS displays the following warning message and creates the `simv` executable:

```
Warning-[IOPCWM] Inout Port connection width mismatch.
Connecting inout ports to
    mismatched width nets has unpredictable results and will
not be permitted in future releases.
    The following 8-bit expression is connected to 16-bit
port "pote" of module "dev", instance "dev1".
Expression: w1
    "expl.v", line_number
```

Using Lint

The `+lint` compile-time option displays lint messages. These messages help you to write very clean Verilog code. The following is an example of a lint message:

```
Lint-[GCWM]    Gate connection width mismatch
```

VCS displays this message when you attach an entire vector reg instead of a bit-select to an input terminal of a gate. In this message the text string `GCWM` is the ID of the message. You can use the ID to enable or disable the message.

The syntax of the `+lint` option is as follows:

```
+lint=[no] ID|none|all, . . .
```

Here:

`no`

Specifies disabling lint messages that have the ID that follows. There is no space between the keyword `no` and the ID.

`none`

Specifies disabling all lint messages. IDs that follow in a comma separated list are exceptions.

`all`

Specifies enabling all lint messages. IDs that follow preceded by the keyword `no` in a comma separated list are exceptions.

The following examples show how to use this option:

- Enable all lint messages except the message with the GCWM ID:

```
+lint=all,noGCWM
```

- Enable the lint message with the NCEID ID:

```
+lint=NCEID
```

- Enable the lint messages with the GCWM and NCEID IDs:

```
+lint=GCWM,NCEID
```

- Disable all lint messages. This is the default.

```
+lint=none
```

The syntax of the `+lint` option is very similar to the syntax of the `+warn` option for enabling or disabling warning messages. Another thing these options have in common is that some of their messages have the same ID. This is because when there is a condition in your code that causes VCS to display both a warning and a lint message, the corresponding lint message contains more information than the warning message and can be considered more verbose.

The number of possible lint messages is not large. They are as follows:

```
Lint-[IRIMW] Illegal range in memory word
Lint-[NCEID} Non-constant expression in delay
Lint-[GCWM] Gate connection width mismatch
Lint-[CAWM] Continuous Assignment width mismatch
Lint-[IGSFPG] Illegal gate strength for pull gate
Lint-[TFIPC] Too few instance port connections
Lint-[IPDP] Identifier previously declared as port
Lint-[PCWM] Port connect width mismatch
Lint-[VCDE] Verilog compiler directive encountered
```

Changing Parameter Values From the Command Line

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 1 value so VCS does not display a warning message.

Enabling the Checking

The `-xzcheck` compile-time option 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 globally or 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-41](#)

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 3-16](#).

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 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 5 *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 example.v -xzcheck nofalseneg
```

If you compile and simulate `example1` or `example2` with the `-xzcheck` compile-time 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 `example1` or `example2` with the `-xzcheck` compile-time option and the `nofalseneg` argument, VCS does not display the warning message.

HSOPT Technology

The HSOPT technology improves both the compile-time and runtime performance of VCS, reduces the amount of memory needed for both compilation and simulation, and reduces the size of the `simv` executable.

HSOPT is an LCA feature requiring a special license.

You enable HSOPT with the `-hsopt` compile-time option.

Any difference in simulation results could be merely a matter of a different order of output messages, however they could also be the result of race conditions.

The designs that benefit the most from HSOPT are as follows:

- 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
- designs compiled with `-debug_all`

The designs that benefit the least from HSOPT are as follows:

- shallow designs — those with only a few layers of hierarchy
- designs without extensive use of timing

HSOPT is developed for Verilog and SystemVerilog code (design, assertion, and testbench constructs) and supports the following adjacent technologies:

- mixed HDL (VCS MX)
- OpenVera Native Testbench
- OpenVera Assertions
- AMS (analog mixed-signal)
- 64 bit compilation and simulation
- all types of coverage
- SystemC cosimulation
- Vera

Making Accessing an Out of Range Bit an Error Condition

By default it is a warning condition if your code assigns a value to, or reads the value of, a bit of a signal, or an element in a memory or multidimensional array, that was not in the declaration of the signal, memory, or array.

For example:

```
reg [1:0] r1;
:
initial
r1[2] = 1;
```

In this case, there is no bit 2 in the declaration of reg `r1`. VCS displays a warning message but continues to compile the code and link together the `simv` executable. The following is an example of this warning message:

```
Warning-[SIOB] Select index out of bounds
in module module_name
   "source_file.v", line_number: signal[bit]
```

You can use the `+vcs+boundscheck` compile-time option to tell VCS to make accessing a bit or element that is outside the declared range to be an error condition, so that VCS does not create the new `simv` executable.

The following is an example of the error message VCS displays when you enter this option and access an undeclared bit:

```
Error-[SIOB] Select index out of bounds
in module module_name
```



```
"source_file.v", line_number: signal[bit]
```

Like the warning message, the error message includes the module name where the access occurs, the source file name, the line number, and the signal name and bit that is outside the declared range, or the memory or array and the undeclared element.

If you access an element that is not in the declared range of a memory or a multidimensional array, and include the `+vcs+boundscheck` compile-time option, VCS displays the previous error message as well as the following error message:

```
Error - [IRIMW] Illegal range in memory word
        Illegal range in memory word shown below
        "source_file.v", line_number: memory[element]
        [element]...
```

Compiling Runtime Options Into the `simv` Executable

You can enter some runtime options on the `vcs` command line or in the file that you specify with the `-f` or `-F` compile-time option and VCS compiles these runtime options into the `simv` executable so you do not need to specify them at runtime.

The runtime options that you can simply enter on the `vcs` command line or in the file that you specify with the `-f` or `-F` compile-time options are as follows:

<code>+cliecho</code>	<code>+no_pulse_msg</code>
<code>+sdverbose</code>	<code>+vcs+finish</code>
<code>+vcs+flush+all</code>	<code>+vcs+flush+dump</code>
<code>+vcs+flush+fopen</code>	<code>+vcs+flush+log</code>

You can also enter the following runtime options on the `vcs` command line or in the file that you specify with the `-f` or `-F` compile-time option, so that VCS compiles them into the `simv` executable, BUT you must precede them with the `+plusarg_save` compile-time option:

<code>+cfgfile</code>	<code>+override_model_delays</code>
<code>+vcs+dumpoff</code>	<code>+vcs+dumpon</code>
<code>+vcs+dumpvarsoff</code>	<code>+vcs+grwavesoff</code>
<code>+vcs+ignorestop</code>	<code>+vcs+learn+pli</code>
<code>+vcs+mipd+noalias</code>	<code>+vcs+nostdout</code>
<code>+vcs+stop</code>	<code>+vera_load</code>
<code>+vera_mload</code>	<code>+vpdbufsize</code>
<code>+vpddrivers</code>	<code>+vpdfile</code>
<code>+vpdfilesize</code>	<code>+vpdnocompress</code>
<code>+vpdnostrengths</code>	<code>+vpdports</code>
<code>+vpdupdate</code>	

You can also include the `-i` runtime option, for specifying a file containing CLI commands, on the `vcs` command line to be compiled into the executable if it is preceded by the `+plusarg_save` option. However, you cannot enter this runtime option in the file that you specify with the `-f` or `-F` compile-time option.

You can also include the `-s` runtime option, to stop simulation as soon as it starts, on the `vcs` command line to be compiled into the executable, without the `+plusarg_save` option, but you cannot enter this runtime option in the file that you specify with the `-f` or `-F` compile-time option.

Several runtime options, such as `-cm`, `-cm_dir`, `-cm_name`, `+notimingcheck`, and `+no_tchk_msg`, are also compile-time options. When these options appear on the `vcs` command line or in the file that you specify with the `-f` or `-F` compile-time option, even if you precede them with the `+plusarg_save` option, VCS considers them to be compile-time options. So, there is no way to compile these runtime options into the `simv` executable.

For more details on using these runtime options, see

Performance Considerations

When you compile your design there are a number of practices that can slow down or speed up both compilation and simulation. This section describes the practices that can speed up both compilation and simulation and then lists the compile-time options that can significantly impede or accelerate compilation and simulation.

Using Local Disks

Disk I/O can dominate VCS compile times, so be sure to use a disk local to the CPU on which you are running the compilation for all permanent and intermediate storage. It is most important that the `csrc` temporary working directory be on a local disk. Use the `-Mdir` compile-time option to specify an alternate `csrc` directory on a local disk of the host machine in the form: `-Mdir=directory`

If the `csrc` working directory is not located on a local disk, set the incremental compile directory so that it is. Or change to the local disk (`cd local_disk`) and run the compilation there. This ensures that VCS can quickly access the compilation directory. If the link step takes more than a few seconds on a small design, then you know that you're accessing files over the network.

Managing Temporary Disk Space on UNIX

The temporary disk space partition (`/tmp`) causes errors if it becomes full. Two major users of this disk space during a Verilog compile are VCS and the C compiler. Either of these can cause large amounts of data to be written to the temporary disk space. Solutions to the problem are as follows:

- The solution to the C compiler's use of temporary disk space is to use the current directory, or other large disk. This is done by adding the appropriate arguments to the C compiler via the VCS compile-time argument `-CC`, as in the following examples for Sun's C compiler on SPARC:

```
vcs -CC "-temp=." a.v
vcs -CC "-temp=/bigdisk" a.v
```

- You can also set the `TMPDIR` environment variable to a large disk location that is different from `/tmp`.

On Sun SPARC, and most other machines that use a C compiler other than Sun's C compiler, the environment variable `TMPDIR` is used to specify compiler temporary storage.

Compile-Time Options That Impede or Accelerate VCS

There are a number of compile-time options that enhance or reduce compilation and simulation performance. Consider looking for opportunities when you can use the options that speed up compilation or simulation and looking for ways to avoid the options that slow down compilation or simulation.

Compile-Time Options That Slow Down Both Compilation and Simulation

-debug

Enables line stepping.

+acc

Enables PLI ACC capabilities.

-full64

Compiles in 64-bit mode for 64-bit mode simulation.

Compile-Time Options That Slow Down Simulation

+pathpulse

Enables the `PATHPULSE$` specparam in specify blocks.

+pulse_e

Drives an X value on narrow pulses.

+pulse_r

Filters out narrow pulses.

+pulse_int_e

Drives an X value on pulses narrower than interconnect delays.

+pulse_int_r

Filters out pulses narrower than interconnect delays.

`+spl_read`

Treats output ports as inout ports.

Compile-Time Options That Slow Down Compilation

`-gen_asm`

Generates assembly code instead of directly generating native code. Can increase compilation time up to 20%.

`-gen_c`

Generates C intermediate code instead of directly generating native code. Can increase compilation time as much as 3x.

`-comp64`

Compiles in 64-bit mode for 32-bit simulation.

Compile-Time Options That Slow Down Compilation but Speed Up Simulation

`-O3` or `-O4`

Applies more than the default level of optimizations when generating C code intermediate files and compiling them. Applying more optimizations slows down the C compiler but speeds up simulation.

Compile-Time Options That Speed Up Compilation but Slow Down Simulation

`-O0`

Turns off all optimizations when generating C code intermediate files and compiling them. Turning off optimizations allows the C compiler to finish sooner but the design simulates slower without these optimizations.

-O1

Applies fewer optimizations when generating C code intermediate files and compiling them. Applying fewer optimizations allows the C compiler to finish somewhat sooner but the design simulates somewhat slower without these optimizations.

Compile-Time Options That Speed Up Both Compilation and Simulation

+nospecify

Tells VCS to ignore specify blocks. If you have extensive specify blocks this can increase both compilation and simulation speed.

Compile-Time Options That Speed Up Simulation

+delay_mode_zero

Disables all delays. Can increase simulation speed but your design will simulate differently.

+notimingcheck

Ignores timing check system tasks.

+nbaopt

Removes intra-assignment delays from nonblocking assignment statements

Compiling for Debugging or Performance

You can use the `-Mdir` compile-time option to create different generated file directories, one directory for debugging and another for performance.

For example, for debugging enter the following `vcs` command line:

```
% vcs -Mdir=csrc_debug -debug source.v
```

This command line enables debugging capabilities but also results in a slower simulating `simv` executable. VCS writes the generated files for this `simv` executable in the directory `csrc_debug`.

For faster simulation enter the following `vcs` command line:

```
% vcs -Mdir=csrc_perf source.v
```

This command line results in a faster simulating `simv` executable. VCS writes the generated files for this `simv` executable in the directory `csrc_perf`.

64-32-Bit Cross-Compilation and Full 64-Bit Compilation

Compile and simulate using less than 4 GB of RAM on a 32-bit machine. However, if you are simulating a very large design, you may need more than 4 GB of RAM. If this is the case, you can access more memory by using either a 64-32-bit cross-compilation or full 64-bit compilation process.

VCS provides two types of compilation processes that take advantage of the additional memory capacity of 64-bit machines:

- **64-32-Bit Cross-Compilation** — In this process you use the `-comp64` option to compile a design on a 64-bit machine; then run the resulting `simv` on either a 32-bit or 64-bit machine.
- **Full 64-Bit Compilation** — In this process, you use the `-full64` option to analyze and compile a design on a 64-bit machine; then run the resulting `simv` on a 64-bit machine.

Note:

- 64-bit machines have more capacity than 32-bit machines, but there is a performance trade-off.
- The process of compiling on a 64-bit machine and simulating the executable on a 32-bit machine, generally known as Cross-compilation is not yet supported.
- The 64-bit compilation and 64-32-bit cross-compilation processes are not available on all platforms, and not all features and capabilities of VCS work with them. Specific requirements can change with each release so check this information in the *VCS Release Notes* (\$VCS_HOME/Doc/ReleaseNotes).
- You can include PLI code in either the 64-32-bit cross-compilation or full 64-bit compilation processes. PLI code compiled with gcc usually works fine without any restrictions. Do not forget to compile the PLI code in 64-bit mode to use it with VCS MX for 64-bit compilation.

Identifying the Source of Memory Consumption

Before running a 64-32-bit cross-compilation or full 64-bit compilation process, there are some steps you can take to identify and possibly solve a memory consumption problem.

If VCS encounters a memory problem during compile-time or runtime, it typically returns one of the following error messages:

```
Error: out of virtual memory (swap space)
(v2ssl_16384x64cm16_func.v line 1053458)
Error: malloc(1937887600) returned 0: Not enough space
```

```
Doing SDF annotation .....
```

```
Error: malloc(400) returned 0: Not enough space
```

```
error: out of virtual memory (swap space)  
error: calloc(16384,1) returned 0: Cannot allocate memory
```

```
Error-[NOMEM] Out of virtual memory (swap space)!  
sbrk(0) returned      0x80f52ea0  
datasize limit  2097148 KB  
memorysize limit      2097152 KB
```

If you encounter one of these error messages, there are several alternative methods you can try that might help adapt to the memory requirements of your design. These methods, described briefly in the next section, apply to cases in which you are simulating large, flat, gate-level designs with no timing or timing checks. If this is not the case, then you should proceed to the methodology described in [“Running a 64-Bit Compilation and Simulation” on page 3-34](#).

Minimizing Memory Consumption

The following list contains several ways you can minimize your compile-time and runtime memory consumption without using the `-full64` or `-comp64` options:

- Check the physical memory on your workstation. You can use the `top` utility for this. All current operating systems, including Linux, support 4 GB of memory. Make sure you are using a machine with enough available memory (that is, even though your machine may appear to have plenty of memory, if there are other processes running concurrently on that machine, you won't have access to the entire memory).

- Avoid using debug options like `-debug`, `-debug_all...` switches.
- In the `pli.tab` files, beware of ACC calls that call for global access (i.e. `acc=rw:*`)
- Minimize the amount of dumping. Instead of dumping the whole design, try to limit the scope of dumping to particular modules. Note that an overhead is incurred if you compile in dumping using `$test$plusargs`, even if it is not enabled until runtime.
- If dumping to a VPD file, use `+nocelldefinepli+n` to limit dumping to non-library modules.

Contact vcs_support@synopsys.com for details on using these methods.

Running a 64-32-Bit Cross-Compilation

If you continue to encounter compile-time memory consumption issues, even after trying the previous methods, then your next step is to try the 64-32-bit cross-compilation process. This process compiles a design using a 64-bit address range, which allows the compilation process to go beyond the 4 GB limit of a 32-bit application. This process produces a `simv` that is a 32-bit executable, which enables you to use your existing 32-bit PLI code and third-party applications during the simulation.

Setting up the Compiler and Linker

Before running the 64-32-bit cross-compilation, Synopsys recommends that you check the *VCS Release Notes* for currently supported compilers and linkers. In general, you can use `gcc` for compiling. The release notes also indicate the required library and assembler patches.

Memory Setup

In order to run the 64-32-bit cross-compilation process, make sure you are running on a machine with at least 8 GB of available memory. The 8 GB can comprise available physical memory plus available swap space.

You can check for the amount of available physical memory and swap space by running the `top` utility, as shown in the following example:

```
% top

% 0 processes:  58 sleeping, 1 running, 1 on cpu
CPU states: 95.2% idle,  0.2% user,  4.6% kernel,  0.0%
iowait,  0.0% swap
Memory: 512M real, 294M free, 60M swap in use, 1333M swap free
```

In general, the amount of swap space should be at least 2.5 times the amount of physical memory. The more the entire process can run using physical memory, the less swapping will occur, giving better overall performance.

If you encounter memory issues, try changing the system limits to values similar to the following example:

```
UNIX> datasize 3070000
UNIX> stacksize 200000
```

Note that these are only experimental values and you may need to further adjust them to fit your particular situation.

If you still have memory issues, try running the cross-compilation process with the `+memopt` option.

Specifying the Compiler, Linker, and `-comp64` Option

When running the 64-32-bit cross-compilation process, you can specify the compiler and linker in either of two ways:

- Using the path environment variable.
- Using VCS compile-time options `-cc` and `-ld`.

VCS assumes that the Sun 64-bit linker is located at the following location:

```
/usr/ccs/bin/sparcv9/ld
```

If VCS can't find this linker, it uses a 32-bit linker.

To run the 64-32 bit cross-compilation process, include the `-comp64` option at the command line, as shown in the following example:

```
% vcs -comp64 Verilog_source_files
```

Running a 64-Bit Compilation and Simulation

If you are encountering memory issues at runtime, you can use the `-full64` option. This option compiles a 64-bit binary executable for simulating in 64-bit mode. In this case, you need to use a 64-bit machine at both compile-time and runtime.

Make sure you check the *VCS Release Notes* for all compatible platforms for running a 64-bit compilation.

Note that VCS assumes the Sun 64-bit linker is located at the following location:

```
/usr/ccs/bin/sparcv9/ld
```

If VCS can't find this linker, it uses a 32-bit linker.

The following example shows how to compile a 64-bit simulation:

```
% vcs -full64 source_files <other compile time options>
```

Using Radiant Technology

VCS's Radiant Technology applies performance optimizations to your design while VCS compiles your source code. These Radiant optimizations improve the simulation performance of all types of designs from behavioral and RTL to gate-level designs. 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 Radiant Technology

You specify Radiant Technology optimizations at compile time. Radiant Technology has the following compile-time options:

`+rad`

Specifies using Radiant Technology

`+optconfigfile`

Specifies applying Radiant Technology optimizations to part of the design using a configuration file. See [“Applying Radiant Technology to Parts of the Design”](#) on page 3-36.

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:

- Backannotating SDF Files

You cannot use Radiant Technology if your design backannotates 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.

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};
```

Here:

`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:
{ }

`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 `;`.

The Verilog comment characters `/* comment */` and `// comment` also work 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 of two module statements, 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.rega} {noOpt};
```

In this example, the module statement disables Radiant optimizations for all instances of module mod1.

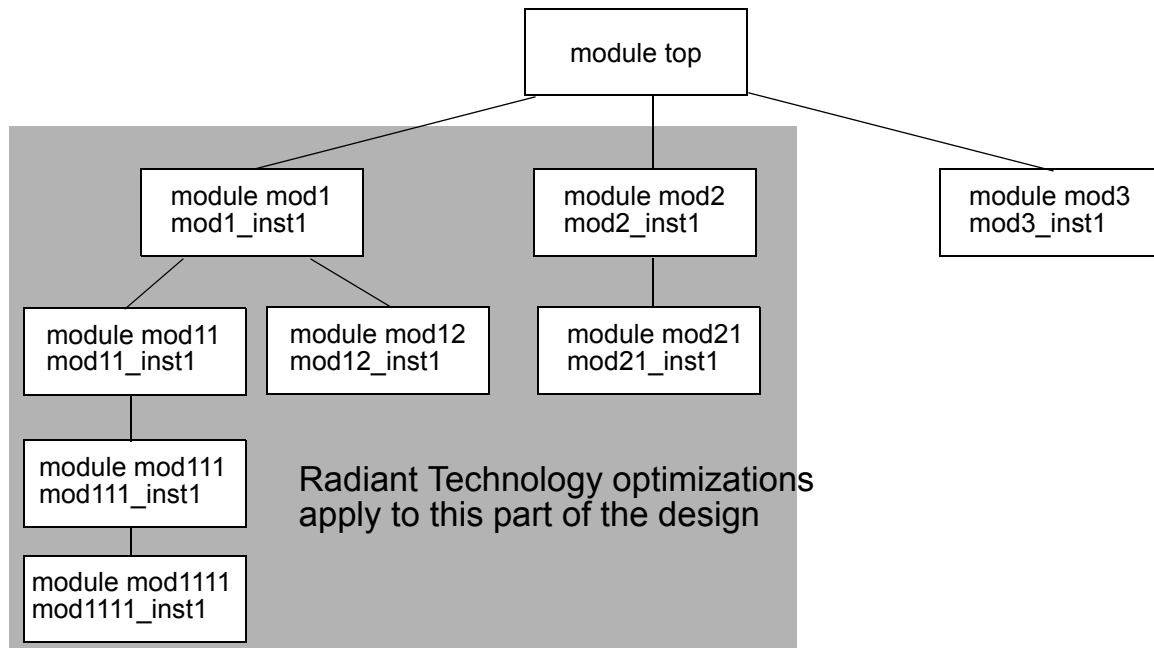
The instance statement disables Radiant optimizations for the following:

- 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 rega in module instance mod2_inst1.

first tree statement example

```
tree {mod1,mod2} {Opt};
```

This example is for a design with the following module hierarchy:



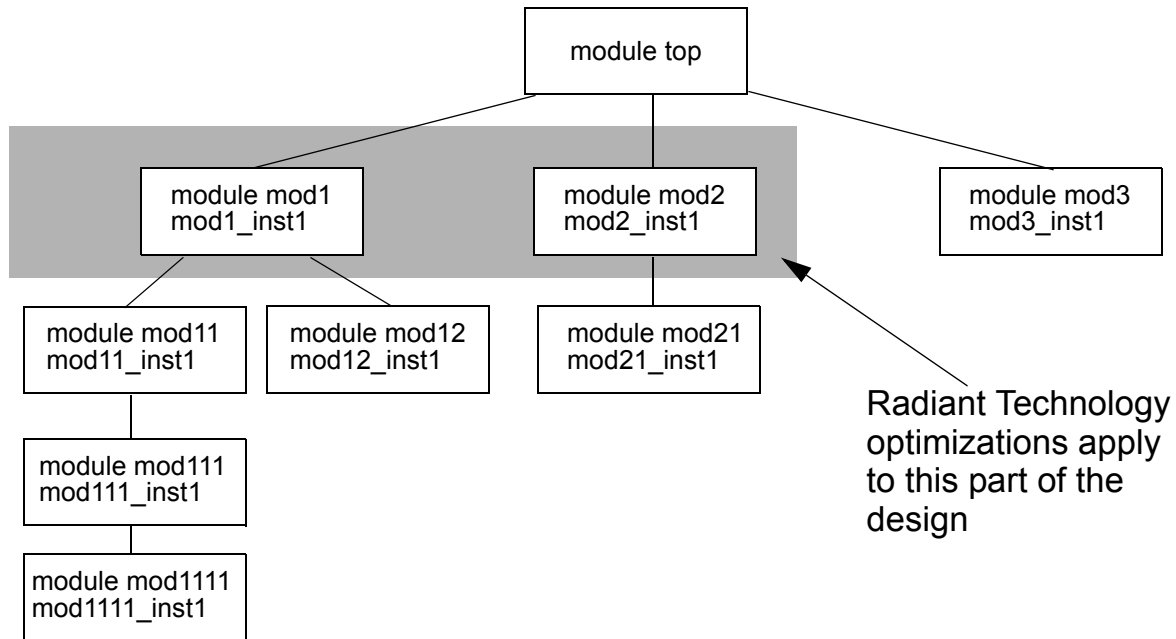
The statement enables Radiant Technology optimizations for the instances of modules mod1 and mod2 and for all the module instances hierarchically under these instances.

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.

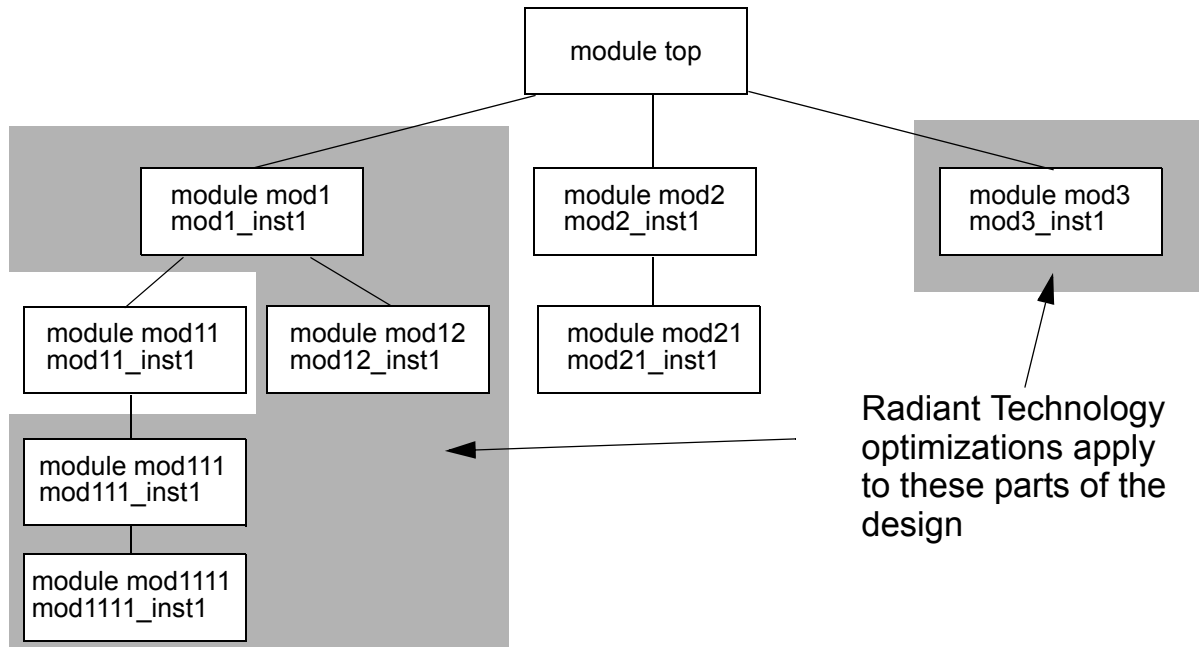
third tree statement example

You can specify a negative value for the depth value. When you do, you 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.

Library Mapping Files and Configurations

Library mapping and configurations are an LCA (Limited customer Availability) feature and requires a special license. For more information contact your Synopsys Applications Consultant.

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 a 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` compile-time option. Using a library mapping file is a Verilog 2001 feature so when you use it you must also include the `+v2k` or `-sverilog` compile-time option.

The paths to the Verilog source files in these logical libraries must also be included on the vcs command line. For example:

```
vcs +v2k dev.v -libmap lib1.map /net/design1/design1_1/  
indev.v /net/design1/design1_2/indev.v
```

Both the `/net/design1/design1_1` and the `/net/design1/design1_2` directories contain a file named `indev.v` for a module named `indev`, but VCS uses the module in `lib1` to resolve instances in `dev.v` and the other module named `indev` is a second and extraneous top-level module.

VCS assigns the source file `dev.v` to the default logical library called `work`.

You can specify either absolute paths or relative paths to the source files.

Overriding the Search Order in the Library Mapping File

You can use the `-liblist logical_library...` compile-time option to alter the search order VCS uses. If the library mapping file lists the `lib1` logical library first, you can tell VCS to search `lib2` first with the following VCS command line:

```
vcs +v2k dev.v -libmap lib1.map -new_dr /net/design1/  
design1_1/indev.v /net/design1/design1_2/indev.v  
-liblist lib2+lib1
```

Specifying Multiple Library Mapping Files

You can enter multiple `-libmap` options for multiple library mapping files. If you do the search order will be based on the order of the logical libraries in these files and the order in which you enter the files on the `vcs` command line.

Displaying Library Matching

You can tell VCS to display how it matches source files to logical libraries with the `-libmap_verbose` compile-time option. VCS display messages similar to the following:

```
Mapping file gatelib/indev.v to logical library 'gatelib'
```

Resolving `'include` Compiler Directives

The source file in a logical library might include the `'include` compiler directive. If so, you can include the `-incdir` option on the line in the library mapping file that declares the logical library, for example:

```
library gatelib /net/design1/gatelib/*.v -incdir /net/
design1/spec1lib, /net/design1/spec2lib;
```

Note:

The `+incdir` VCS compile-time option, on the `vcs` command line, overrides the `-incdir` option in the library mapping file.

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, but so is a user-defined primitive. Similarly, a configuration is also a cell. A SystemVerilog interface and testbench program block are also types of cells. A configuration uses the concept of a cell and in fact a `cell` is a keyword in a configuration.

Configurations do the following:

- Specify a library search order for resolving cell instances (so can 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.

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 `rtlLib` and you had gate-level code describing the adder in a logical library named `gatelib`. Now for some reason, you want 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 would be something like this:

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

Now you are going to instantiate this eight-bit adder eight times to make a 64 bit adder. You want to use configuration `cfg1` for the first instance of the eight-bit adder but not in any other instance. A configuration to do so would be 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 -new_dr ...
vcs -top test -sverilog -new_dr ...
```


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

4

Simulating Your Design

You can simulate your design with VCS using several options and techniques, which allow you to focus on either performance or debugging. You can also use runtime options to save and restart the simulation as required.

This chapter covers the following topics:

- [Running and Controlling a Simulation](#)
- [Save and Restart](#)
- [Specifying a Very Long Time Before Stopping Simulation](#)
- [Passing Values From the Runtime Command Line](#)
- [How VCS Prevents Time 0 Race Conditions](#)
- [Improving Performance](#)
- [Profiling the Simulation](#)

Running and Controlling a Simulation

This section describes how to simulate your design using the binary executable generated during compilation.

Invoking a Simulation at the Command Line

To invoke the simulation, enter the following at the command line:

```
% executable [options]
```

The options are more than one runtime options that enable you to control how VCS executes the simulation. For a complete list of VHDL and Verilog runtime options, see [Appendix C, "Simulation Options"](#).

You control and monitor the simulation by using UCLI, CLI or SCL commands. Detailed descriptions of these commands are available in [Chapter 8, "Unified Command-Line Interface \(UCLI\)"](#) and [Chapter 9, "Using the Old Command Line Interface \(CLI\)"](#)

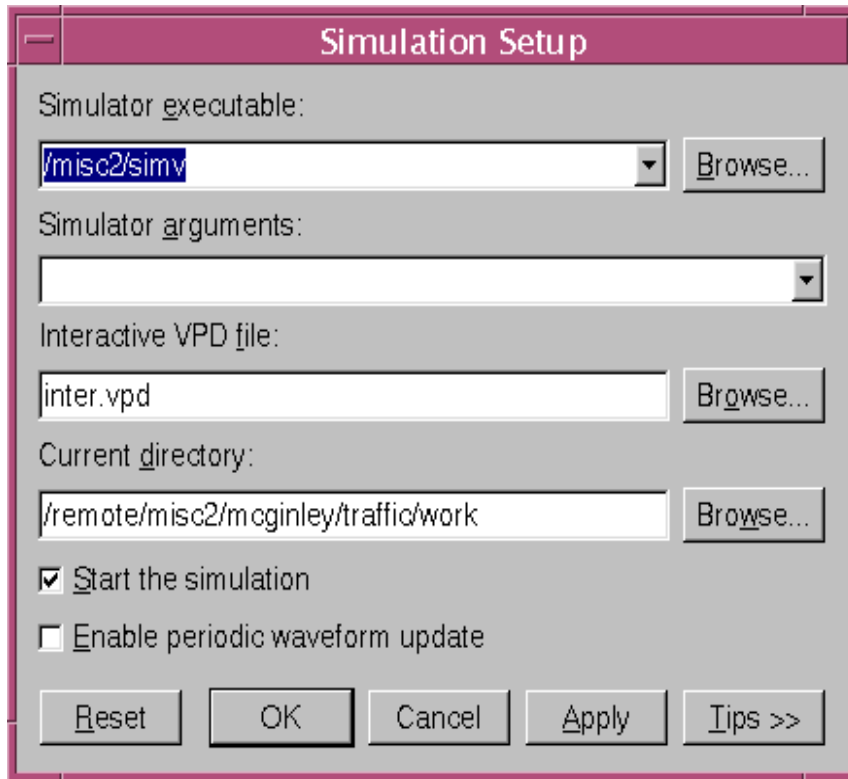
Invoking a Simulation From DVE

To open DVE and start the simulation, do the following:

1. From the command line, open DVE.

```
% dve
```

2. Choose Simulator > Setup, then start simulation from the Simulation Setup dialog box.



3. Browse to the simulation executable or a VPD/VCD file and select a working directory.
4. Select start and waveform update options.
5. Click OK.

For more information on DVE, see [Chapter 5, "Using the Discovery Visual Environment"](#).

Save and Restart

VCS provides a save and restart feature that allows checkpoints of the simulation to be saved 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.

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 4-1 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 4-1 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
```

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 time of save as the old file, then VCS appends further output to that file. Otherwise, VCS attempts to open a file with file name equal to the old file name plus the suffix `.N`. If a file with this name 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 `miscf` callbacks with reasons `reason_save` and `reason_restart`.

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

A more detailed example follows to illustrate this. 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
```

Restarting at the CLI Prompt

The `$restart` system task allows you to restart the simulation at the CLI prompt. Enter it with the name of the check file created by the `$save` system task. For example:

```
C1 > $restart("checkfile1");
```

Specifying a Very Long Time Before Stopping 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 not just halts but ends simulation. This is also with the proviso that the time value be less than 2^{32} .

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

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

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

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

This difference is the first argument.

You can let VCS 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. Round down this quotient to the whole number. This whole number is the second argument.

In this example, you round 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 enter the following runtime option:

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

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
```

How VCS Prevents Time 0 Race Conditions

At simulation time 0, VCS always executes the always blocks in which 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 so the simulator does not execute the always block. Then the simulator executes the initial block and initializes `rst`. When this happens the simulator does not re-evaluate the event control sensitivity list for the always block.

Improving Performance

When you simulate your design you can look for ways to improve the simulation performance. There are runtime options that enable VCS to simulate faster or slower.

Some runtime options enable VCS to simulate your design faster because they allow VCS to skip certain operations. You should consider using these runtime options. They are as follows:

`+vcs+ignorestop`

Tells VCS to ignore the `$stop` system tasks in your source code.

`+notimingcheck`

Disables timing check system tasks. Using this option at compile time results in even faster simulation than using it at runtime.

Runtime options that specify writing to a file slow down simulation. These runtime options are as follows:

`-a filename`

Appends all output of the simulation to the specified file as well as sends it to the standard output.

`-l filename`

Writes all output of the simulation to the specified file as well as to the standard output.

Other runtime options that specify operations other than the default operations also slow down simulation to some extent.

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 shows you 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 4-2 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 shows you 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 4-3 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 hierarchal 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 times 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 shows the simulation time used by the testbench program, the number of instances, and the line number where it starts in its source file.

Example 4-4 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 shows you the CPU time used by different types of Verilog constructs in each module definition in the module view. There are the following 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 4-5 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% 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 shows you 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 4-6 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 that it is 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 4-7 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 4-8 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 shows you 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 4-9 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 shows you 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 4-10 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's 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, 56.66% of the total memory, to simulate the design.

VCS used 890408 bytes of memory, 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 4-11 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 shows the amount of memory used, and the percentage of memory used, by each testbench program.

Example 4-12 Program View

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


5

Using the Discovery Visual Environment

This chapter introduces the Discovery Visual Environment (DVE) graphical user interface. It contains the following sections:

- [Overview of DVE Window Configuration](#)
- [DVE Panes](#)
- [Managing DVE Windows](#)
- [Using the Menu Bar and Toolbar](#)
- [Setting Display Preferences](#)

For complete information on the use of DVE, see the *Discovery Visual Environment User Guide* in your VCS / VCS MX installation.

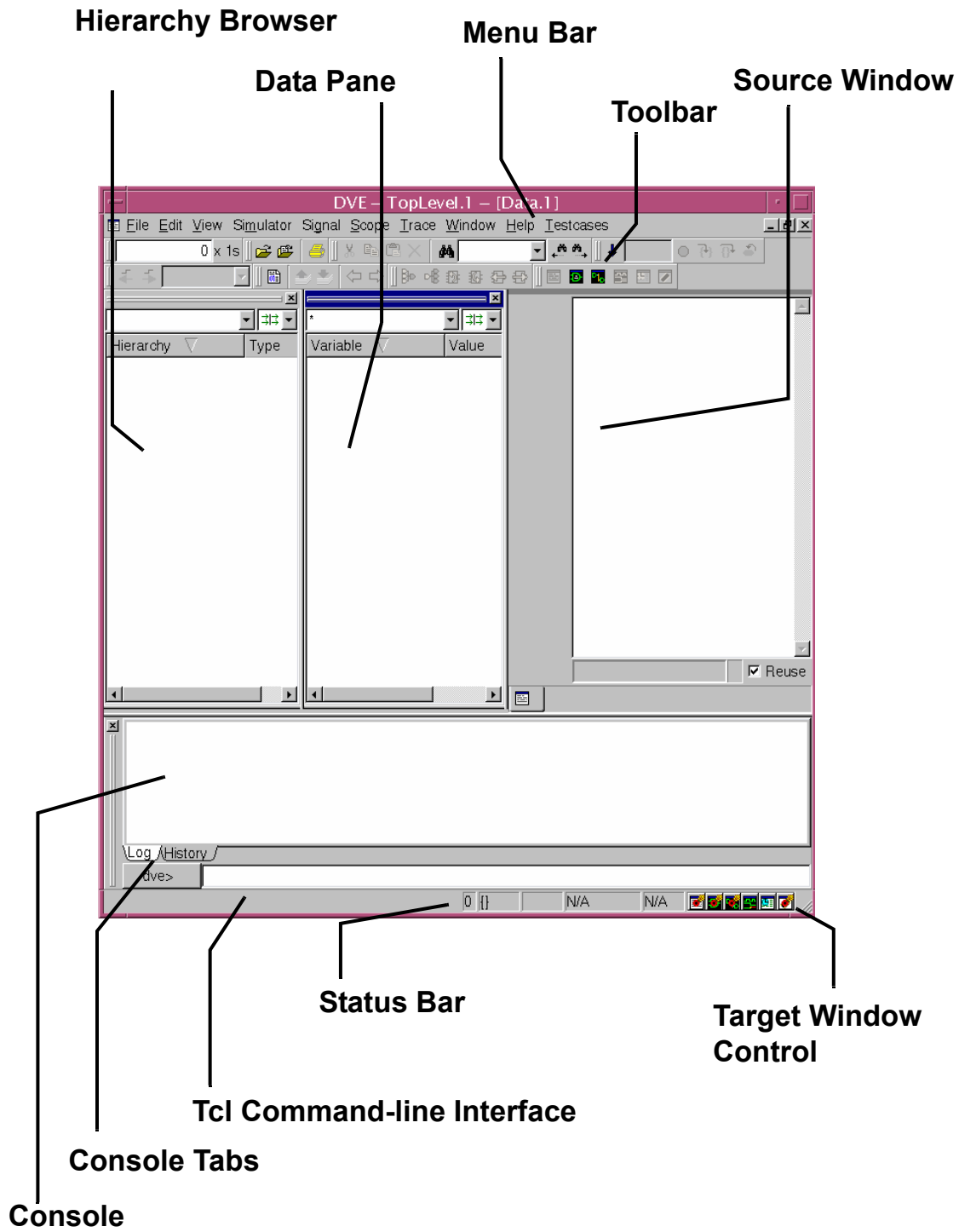
Overview of DVE Window Configuration

DVE has a completely flexible window model. This model is based on the concept of the TopLevel window.

A TopLevel window contains a frame, menus, toolbars, status bar, and pane targets. Any number of TopLevel windows are possible. The default at startup is one.

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 5-1](#) shows the default TopLevel window

Figure 5-1 DVE TopLevel Frame Initial View



DVE Panes

A TopLevel window can contain any number of panes. A pane is a window that serves a specific debug purpose. Examples of panes are Hierarchy, Data, Assertion, Wave, List, Memory, and Schematic.

Panes can be docked on any side of a TopLevel window or left floating in the area in the frame not occupied by docked panes (called the workspace). Panes can also be opened in a new TopLevel frame.

Managing DVE Windows

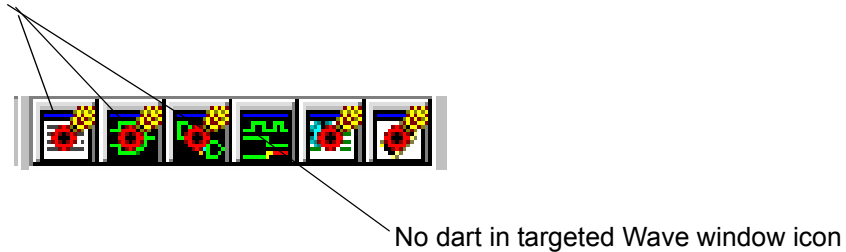
A DVE TopLevel window can contain any number of DVE windows and panes. You can choose to display data in one or many DVE windows and panes by setting defaults, using the status bar window controls, or docking and undocking windows as you work.

Managing Target Panes

The target policy dictates where panes will be created. On each TopLevel at the bottom right corner of the frame are target icons ([Figure 5-2](#)). These icons represent pane types.

Figure 5-2 Window targeting icons

Darts indicate targeted windows are attached to the current window.



Target icons can have the following two states:

- Targeted – Icon has a dart in it, which means an action that requires a new pane creates that pane in the current frame
- Untargeted – icon has no dart in it, which means an action that requires a new pane creates a new TopLevel window that contains that pane.

To open a pane in a new TopLevel window:

1. Click on the icon in the status bar to remove the default dart. .



Targets a new Source pane in a new TopLevel window .

Source



Targets a new Schematic window pane in a new TopLevel window.

Schematic



Targets a new Path Schematic pane in a new TopLevel window.

Path Schematic



Wave

Targets a new Wave pane in a new TopLevel window



List

Targets a List pane in a new TopLevel window.



Memory

Targets a new Memory pane in a new TopLevel window.

2. Click a corresponding window icon in the toolbar to open a window of that type. It will not be attached to the current window and will open in a new TopLevel window.

Docking and Undocking Windows and Panes

You can use the Windows menu to dock and undock windows and panes.

- Select **Windows > Dock in New Row**, then select the row position in which to dock the currently active window.
- Select **Windows > Dock in New Column**, then select the column position in which to dock the currently active window.
- Select **Undock** to detach the currently active window or pane.

General stuff about docked windows such as the hierarchy window.

To delete a window, click the X icon in the corner of the pane. This is the same for all dockable windows.

Dark blue color of dock handle (dock handle is the train track that connects to the X icon) indicates that this docked window is active. This is the same for all dockable windows. An action must occur such as a click to make the window active.

Dragging and Dropping Docked windows

Left Click on the dock handle and drag and drop the window to a new dock location or to a non docked window.

Right click on dock handle brings up a small popup menu:

Undock	Undock the active window.
Dock	Left – Docks the selected window to the left wall of the TopLevel window. Right – Docks the selected window to the right wall of the TopLevel window. Top – Docks the selected window to the top wall of the TopLevel window. Not recommended.. Bottom – Docks the selected window to the bottom wall of the TopLevel window.

Using the Menu Bar and Toolbar

The menu bar and toolbar allow you to perform standard simulation analysis tasks, such as opening and closing a database, moving the waveform to display different simulation times, or viewing HDL source code.

Most items in the menu bar correspond to icons or text fields in the toolbar. For example, you can set the simulation time display in the waveform by doing either of the following:

- Select **View>Go To Time**, then enter a value in the Go To Time dialog box, and click **Apply** or **OK**.
- Enter a value in the Time text field on the toolbar, then press **Enter** on your keyboard.

See [Figure 5-3](#) for an example.

Figure 5-3 Methods for Setting the Simulation Time

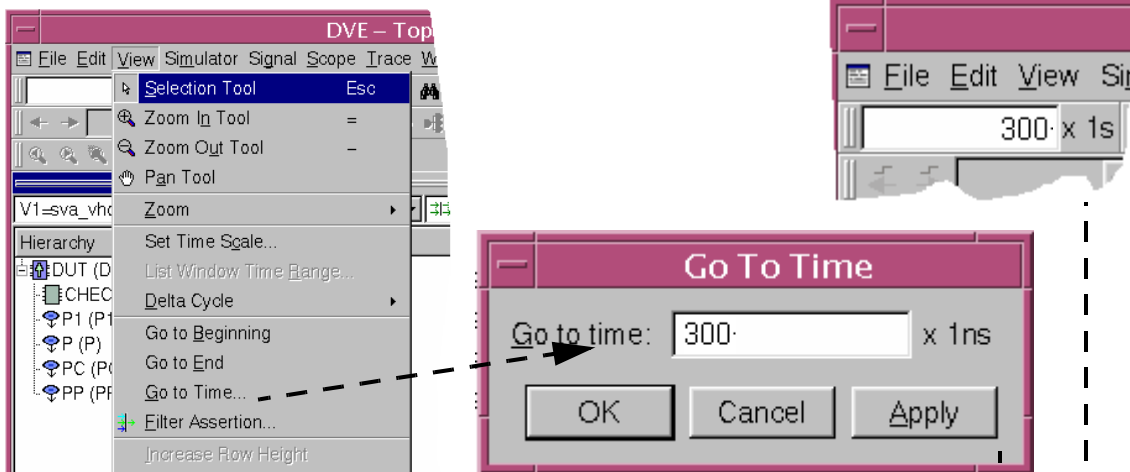
Menu Bar:

OR

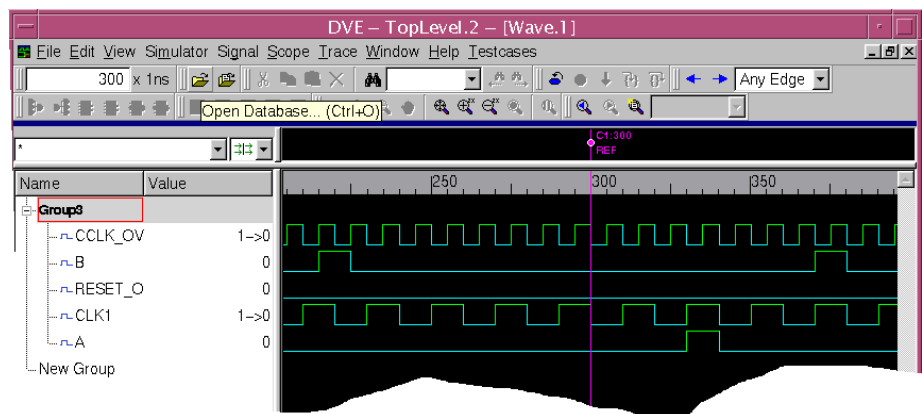
Toolbar:

Select **View>Go To Time**, enter a value in the Go To Time dialog box, then click **Apply** or **OK**.

Enter value in Time text field of the toolbar, then press the Enter key.



Results: Waveform display moves to specified simulation time.



Setting Display Preferences

You can set preferences to customize the display of DVE windows and panes.

To customize the display:

1. In the TopLevel window, select **Edit > Preferences**.

The Application Preferences dialog box displays the Global Settings category.

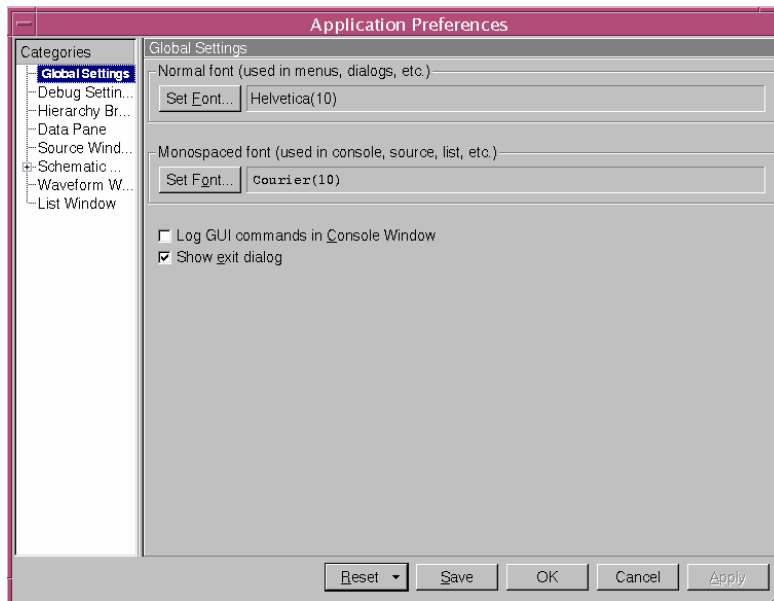
2. Select settings as follows:

- Global Settings

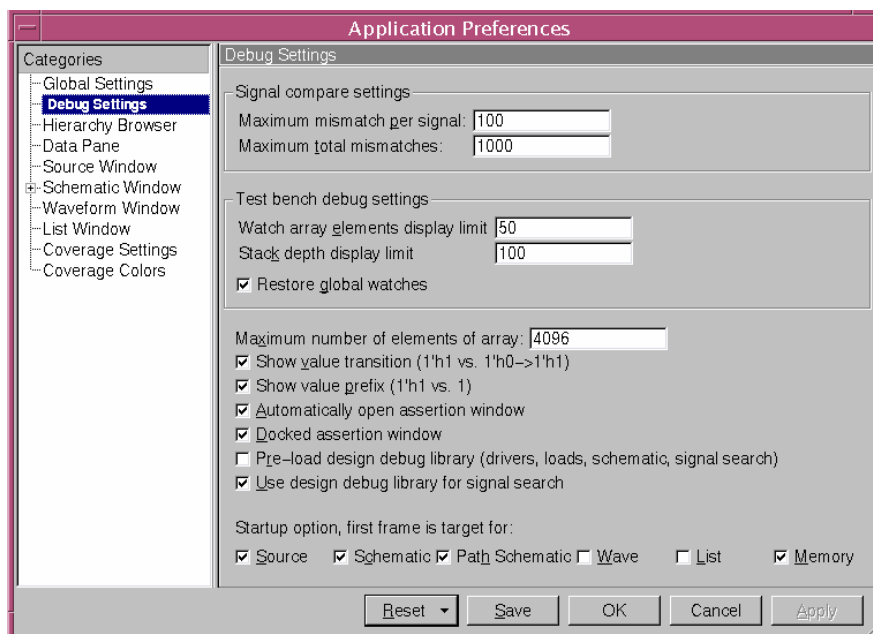
Select settings to set the font and font sizes to display in DVE windows

The default is to log only UCLI commands. To also log GUI commands select the Log GUI commands in Console window checkbox.

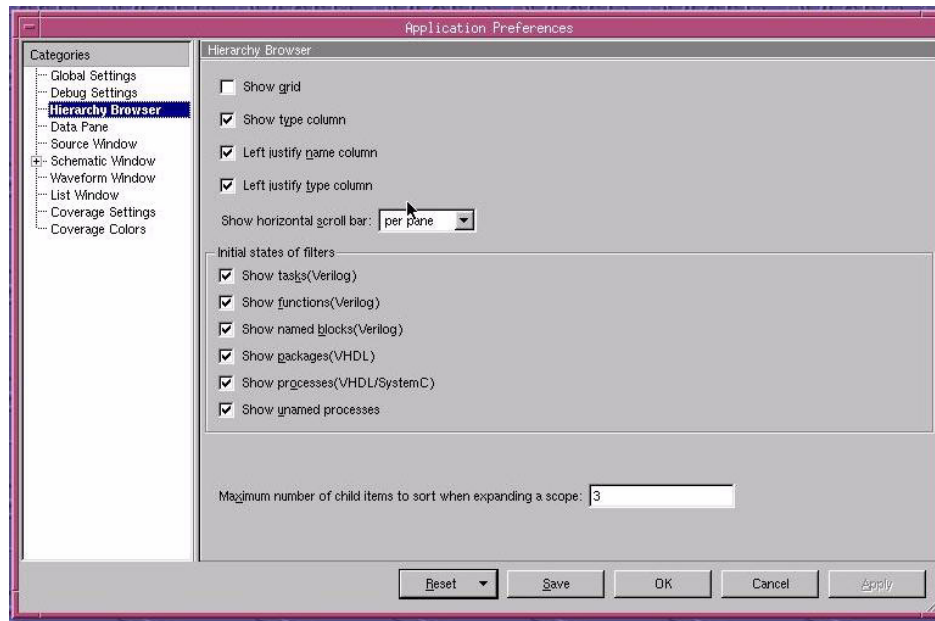
Select whether to display the exit dialog box when closing DVE.



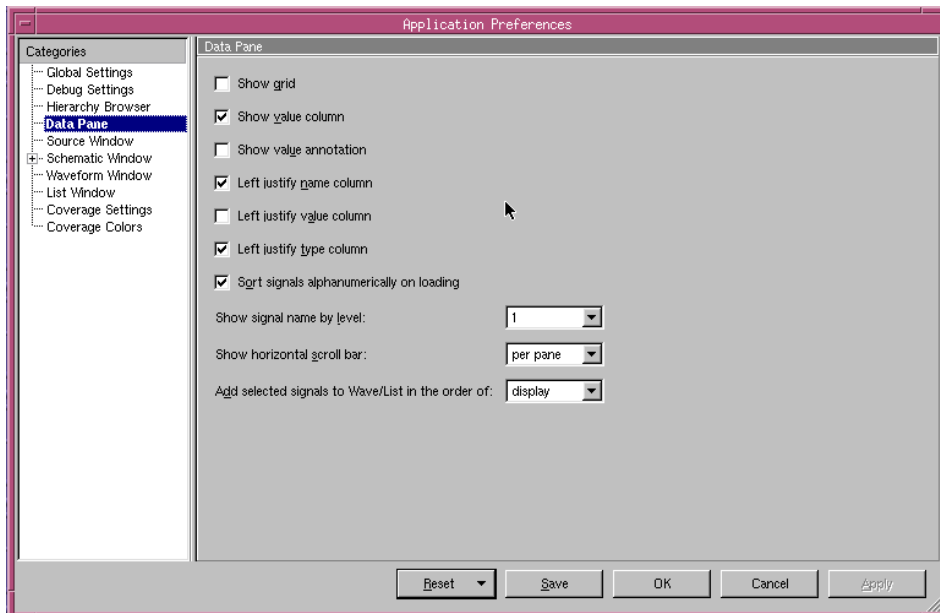
- Debug Settings – Select signal compare parameters, value transition, exit dialog box and assertion window docking defaults, and first frame target setup options..



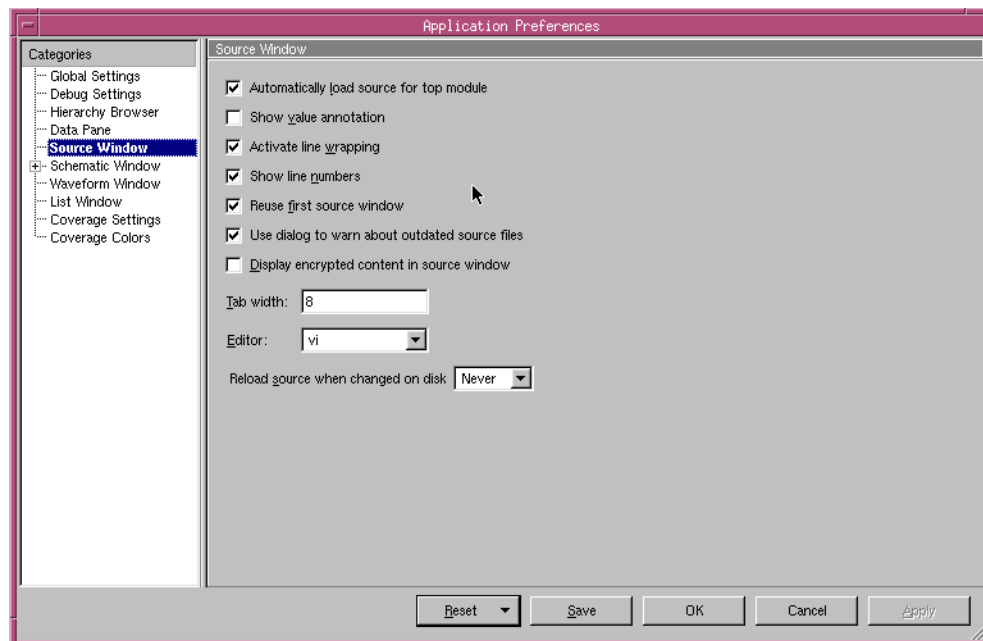
- Hierarchy Browser – Set the appearance and initial filter states..



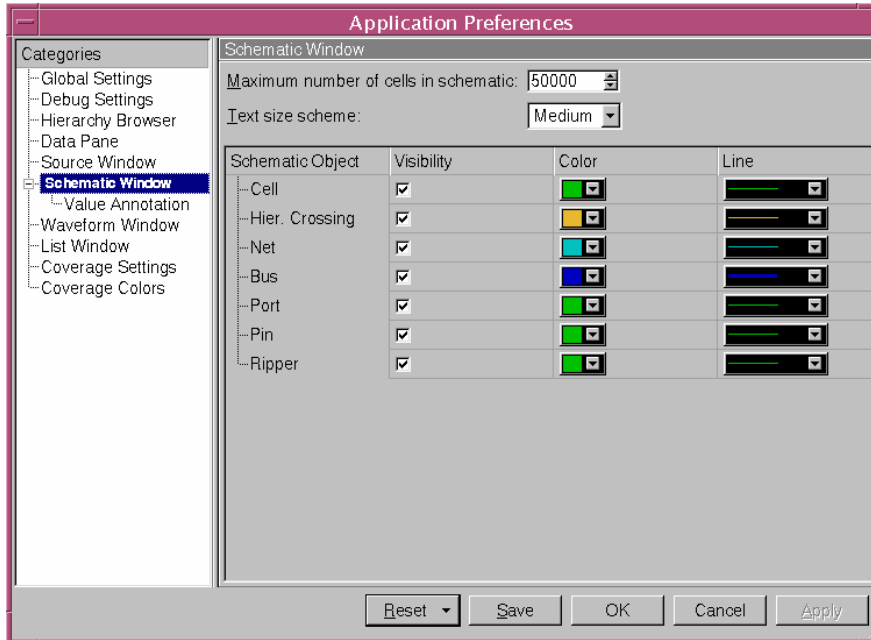
- Data Pane – Set the appearance parameters, signal sorting, signal levels to display, and scroll bar conditions..



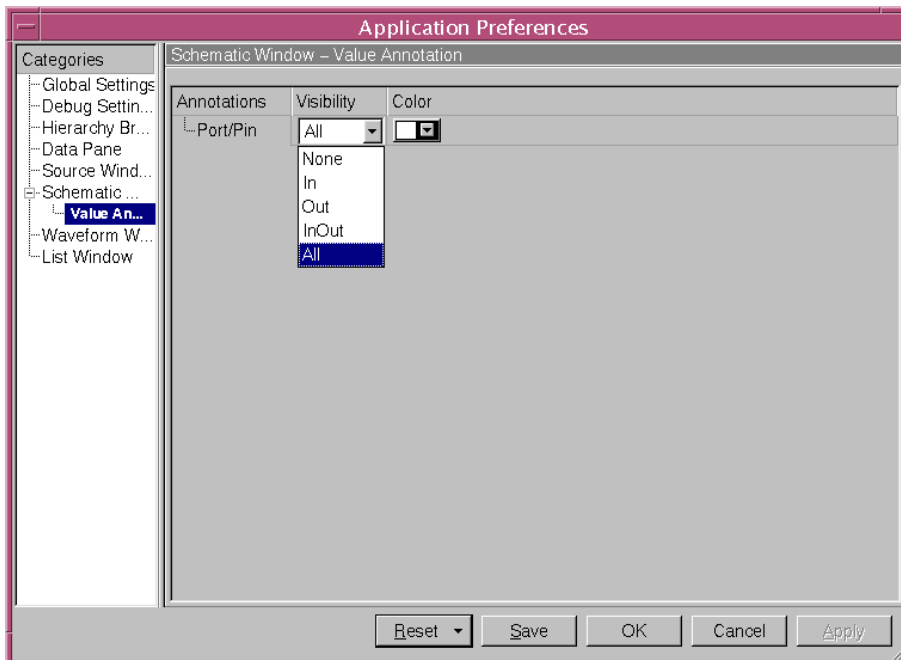
- Source window – Specify data and annotation loading options, line wrap, line number display, tab width, default editor, and automatic reload of changed source code..



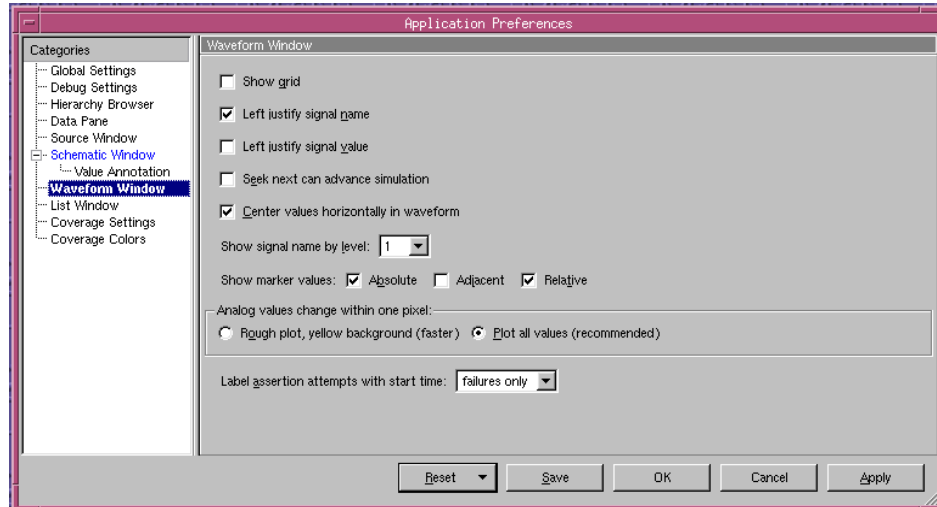
- Schematic window – Set line colors for schematic objects in Schematic and Path Schematic windows..



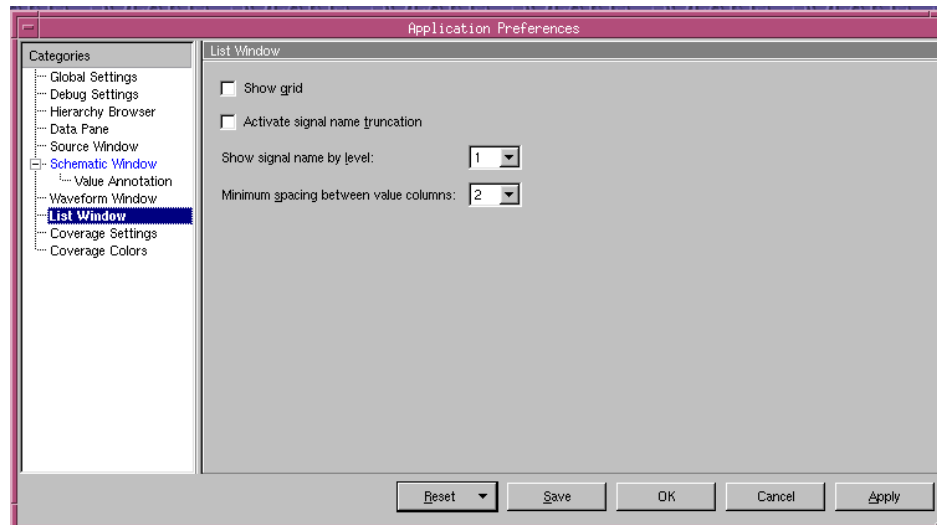
- Select the Value Annotations subcategory and set the Port/Pin visibility and color..



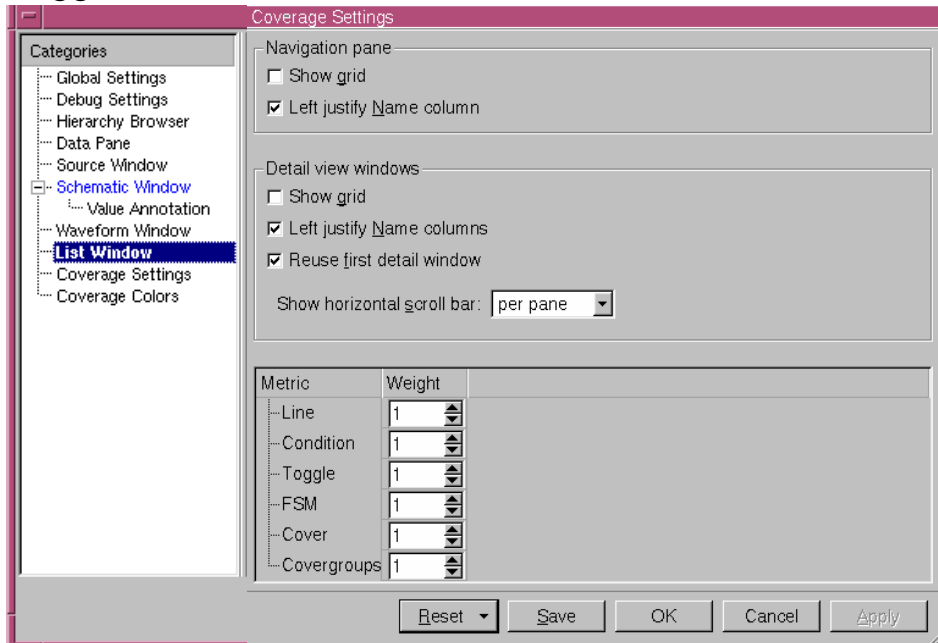
- Waveform window – Set appearance parameters, signal levels to display, and marker value display settings..



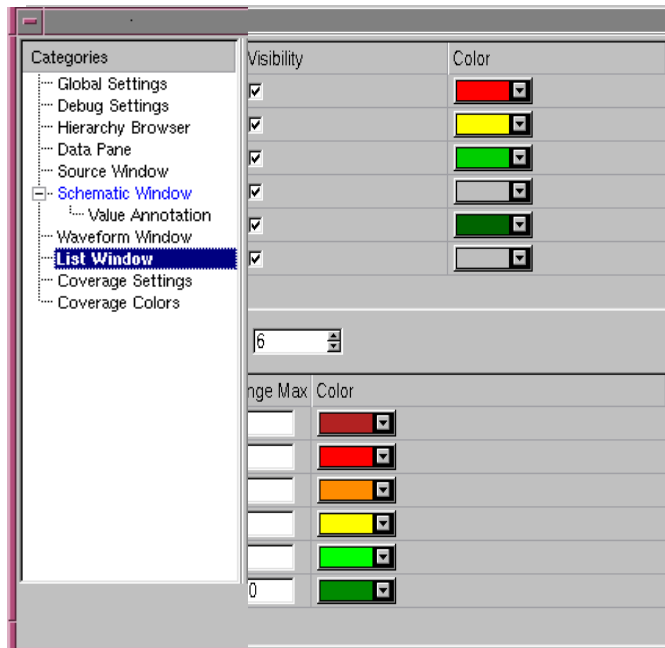
- List window – Specify grid display, signal name truncation, signal levels to display, and column spacing settings..



- Coverage Settings – Set weights for display of line, condition, toggle, FSM, and cover metrics..



- Coverage Colors – Customize the color display of Source window cover states and the number of coverage ranges and their associated colors in the Color Settings window.



3. Click **OK** to save your selections and close the dialog box, **Save** to save your settings and keep the dialog box open, or **Reset** to return the default settings.

6

VPD and EVCD File Generation

VPD and EVCD files contain simulation history data. In Verilog simulation, `$vcdplus` system tasks create these files and name them `vcdplus.vpd` by default.

You can use system tasks that include the `vcdplus` name in the tasks, for example, `$vcdpluson`, `$vcdplusoff`, and `$vcdplusfilename`, to manipulate VPD files. To generate an EVCD file, you can use the system tasks `$dumports`, and `$lsi_dumpports`. You can enter these system tasks in Verilog code or at the Interactive Window command prompt.

You can also use specific runtime options to control how VPD files are generated. These runtime options include `+vpd` in their names, for example, `+vpdbufsize`, `+vpdignore`, etc.

This chapter covers the following topics:

- [Advantages of VPD](#)

- [System Tasks and Functions](#)
- [Runtime Options](#)
- [VPD Methodology](#)
- [EVCD File Generation](#)

Advantages of VPD

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

- Provides a compressed binary format that dramatically reduces file size as compared to VCD and other proprietary file formats.
- The VPD compressed binary format dramatically reduces signal load time.
- Allows data collection for signals or scopes to be turned on and off during a simulation run, thus, dramatically improving simulation run time and file size.
- Can save source statement execution data. This allows instant replay of source execution in the DVE Source Window.

VPD has a set of command line options that affect performance and file sizes and which allow you to run VPD in the most effective manner. 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.

System Tasks and Functions

VPD system tasks capture and save value change data in a binary format so that you can view the data in the Wave Window, Register Window, Source Window, and Logic Browser. You can include the following VPD system tasks in source files or enter them at the DVE interactive prompt.

System Tasks to Generate a VPD File

Note:

The `$vcdpluson` and `$vcdplusoff` system tasks accept the same arguments as the Verilog `$dumpvars` system task. Unlike standard VCD, this lets you turn recording on or off for variables during the same simulation.

`$vcdpluson`

The `$vcdpluson` task begins recording signal value changes of the specified scopes or signals to the VPD history file.

Syntax:

```
$vcdpluson (level, scope*, signal*);
```

Here:

level

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; default is zero).

scope

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

signal

Specifies the name of the signal in which to record signal value changes (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.

```
$vcdpluson;
```

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

```
$vcdpluson(test.risc1.alureg);
```

Example 3: Record two levels of signal value changes: scope (`test`) and one level below.

```
$vcdpluson(2, test);
```

\$vcdplusoff

The `$vcdplusoff` task stops recording the signal value changes for specified scopes or signals.

Syntax:

```
$vcdplusoff (level, scope*, signal*);
```

Example 1: Turn recording off.

```
$vcdplusoff();
```

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

```
$vcdplusoff(test.risc1.alu1);
```

Example 3: Stop recording signal value changes for
`test.risc1.alu1` and `test.risc1.instreg.d1.`

```
$vcdplusoff(test.risc1.alu1, test.risc1.instreg.d1);
```

Example 4: Stop recording signal value changes for scope
`test.risc1.alu1` and 39 levels below. In this example, 40 is a
number large enough to ensure all lower levels are turned off.

```
$vcdplusoff(40, test.risc1.alu1);
```

Note:

The `$vcdpluson/off` commands increment/decrement an internal counter for each signal to be recorded. If multiple `$vcdpluson` commands cause a given signal to be saved, the signal will continue to be saved until an equivalent number of `$vcdplusoff` commands apply to the signal.

`$vcdplusflush`

The `$vcdplusflush` task flushes to the VPD data file any value changes that have been reported by VCS but have not yet been written to the VPD data file.

Syntax:

```
$vcdplusflush;
```

\$vcdplusautoflushon

When simulation stops, the `$vcdplusautoflushon` task automatically flushes to the VPD data file any value changes that have been reported by VCS but have not yet been written to the VPD data file.

Syntax:

```
$vcdplusautoflushon;
```

\$vcdplusautoflushoff

The `$vcdplusautoflushoff` task turns off the automatic flush (enabled by the `$vcdplusautoflushon` task).

Syntax:

```
$vcdplusautoflushoff;
```

\$vcdplusfile

The `$vcdplusfile` task specifies a VPD file name. If it does not specify a name, `vcdplus.vpd` is the default.

Syntax:

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

\$vcdplusclose

The `$vcdplusclose` task terminates all tracing, flushes data to file, closes the current VPD file, and resets all default settings.

Syntax:

```
$vcdplusclose;
```

System Tasks and Functions for Multi-Dimensional Arrays

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 `$vcdplusemon` and `$vcdplusemoff` system tasks, records data each time an MDA has a data change.
- The second method, which uses the `$vcdplusememorydump` system task, stores data only when the task is called.

Syntax for Specifying MDAs

Use the following syntax to specify MDAs using the `$vcdplusemon`, `$vcdplusemoff`, and `$vcdplusememorydump` system tasks:

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

Here:

system_task

Name of the system task (required). It can be `$vcdplusemon`, `$vcdplusemoff`, or `$vcdplusememorydump`.

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 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 Nth dimension.

dimNRsb

This is an optional argument that specifies the right bound of the Nth 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.

Using `$vcdplusemon` and `$vcdplusemoff`

You can use the `$vcdplusemon` and `$vcdplusemoff` tasks to turn on and off, respectively, the recording of changes within memories or MDAs in a design. By using these tasks in VCS, you are able to view changes of memories and MDAs in DVE windows.

Running VCS

In order for VCS to provide MDA data using the `$vcdplusemon` and `$vcdplusemoff` tasks, it requires the `+memcbk` and the `+v2k` switches.

VCS example:

```
vcs -R -I mda.v +memcbk +v2k
```

MDA declaration example:

```
reg [1:0] mem [3:0] [6:4];
```

In order for VCS to provide memory data, it requires the `+memcbk` switch.

VCS example:

```
vcs -R -I mda.v +memcbk
```

Memory declaration example:

```
reg [1:0] mem [3:0];
```

Examples

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

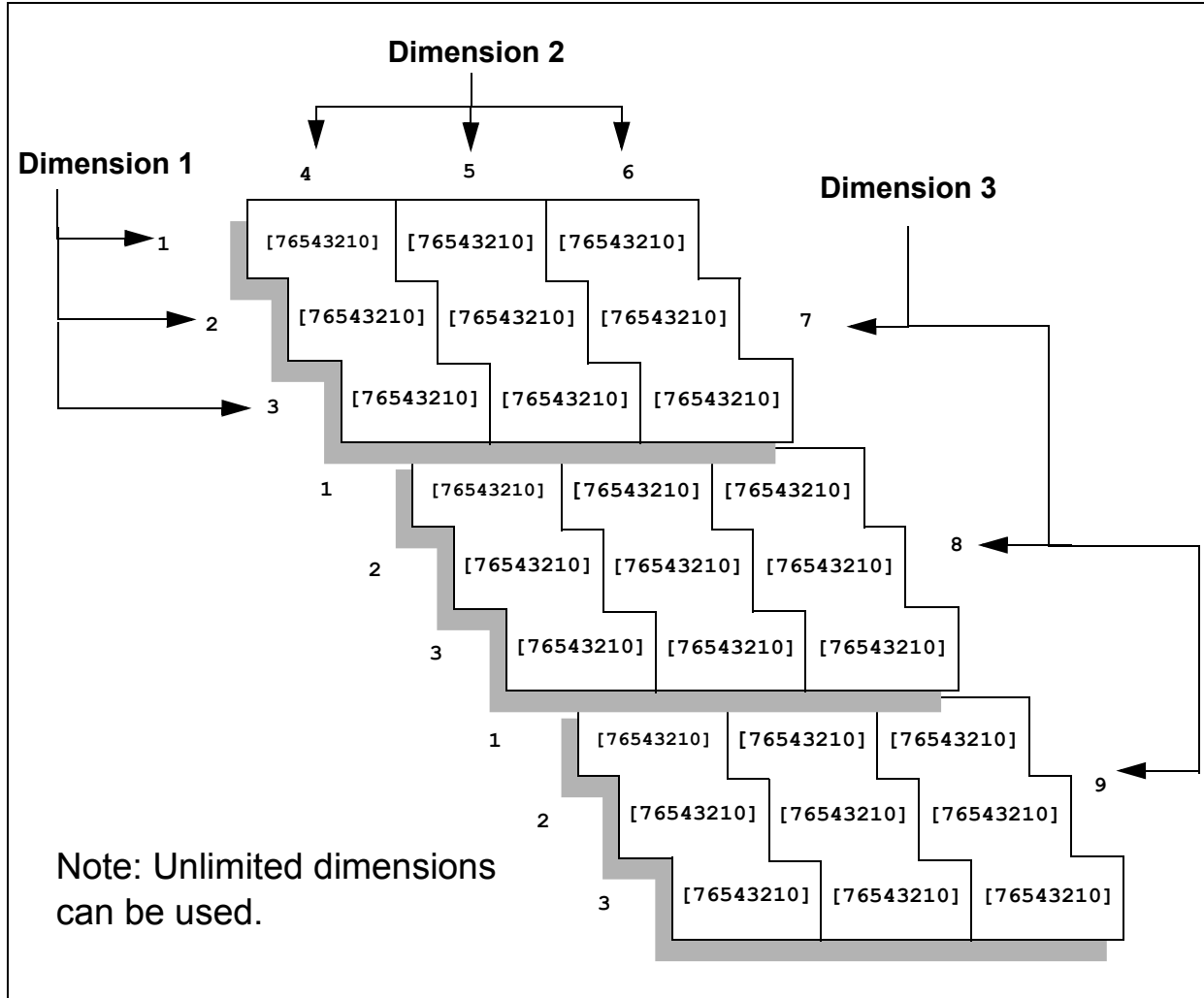
Example 6-1 MDA and Memory Declaration

Note that `mem01` in this example is a three-dimensional array. It has 3x3x3 (27) locations; each location is 8 bits in length.

```
reg [3:0] addr1L, addr1R, addr2L, addr2R, addr3L, addr3R;  
reg [7:0] mem01 [1:3] [4:6] [7:9]
```

See Figure 6-1 for an graphical representation of Example 6-1.

Figure 6-1 Diagram of example: `reg [7:0] mem01 [1:3] [4:6] [7:9]`



Example 6-2 `$vcdplusemon(mem01, addr1L);`

```
$vcdplusemon( mem01 );  
    // Records all elements of mem01 to the VPD file.
```

```
addr1L = 2;  
$vcdplusemon( mem01, addr1L );  
// Records elements mem01[2][4][7] through mem01[2][6][9]
```


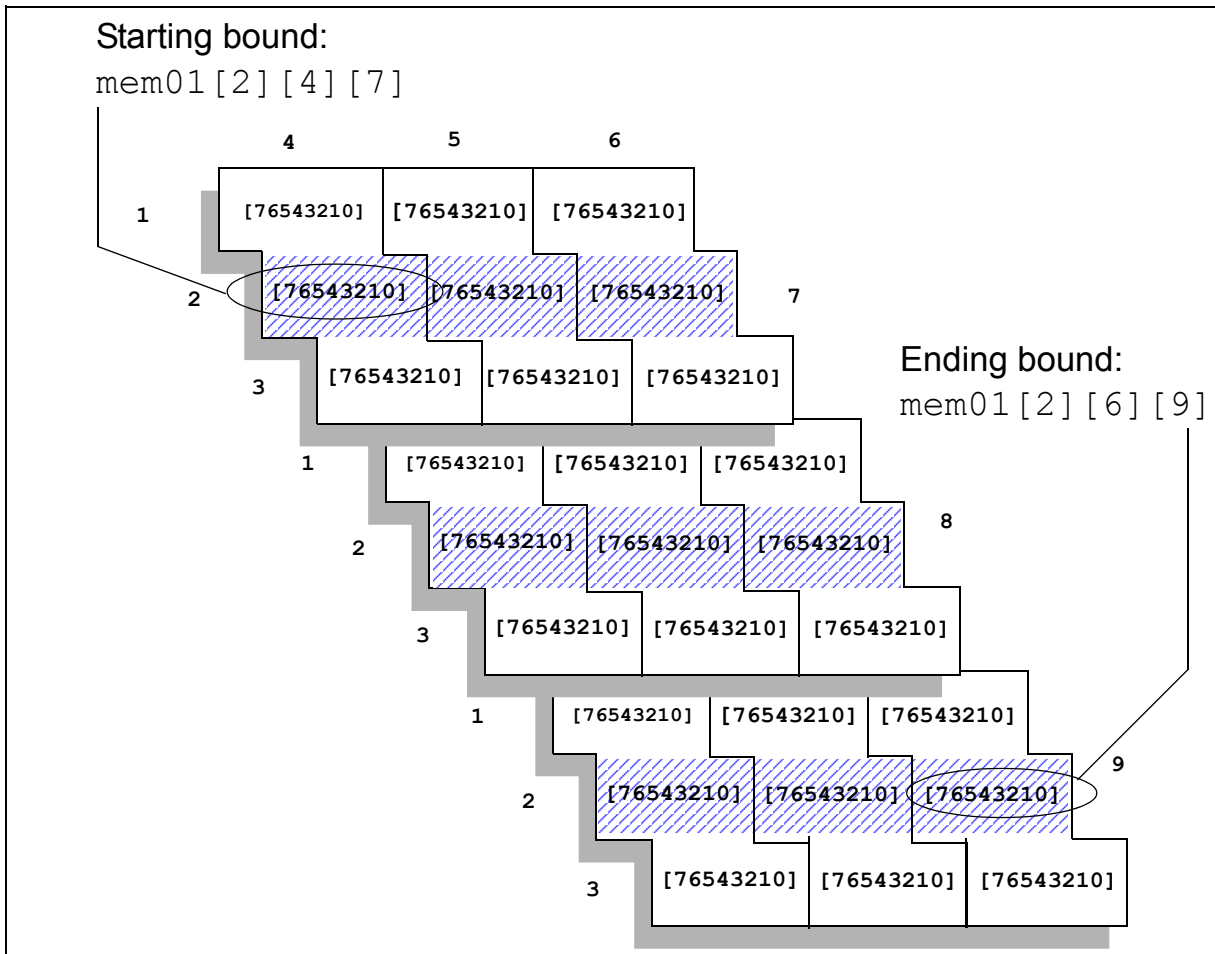
The elements highlighted by the  in the diagram in Figure 6-2 demonstrate Example 6-2.

Figure 6-2 Diagram of example: `$vcdplusemon(mem01, addr1L);`

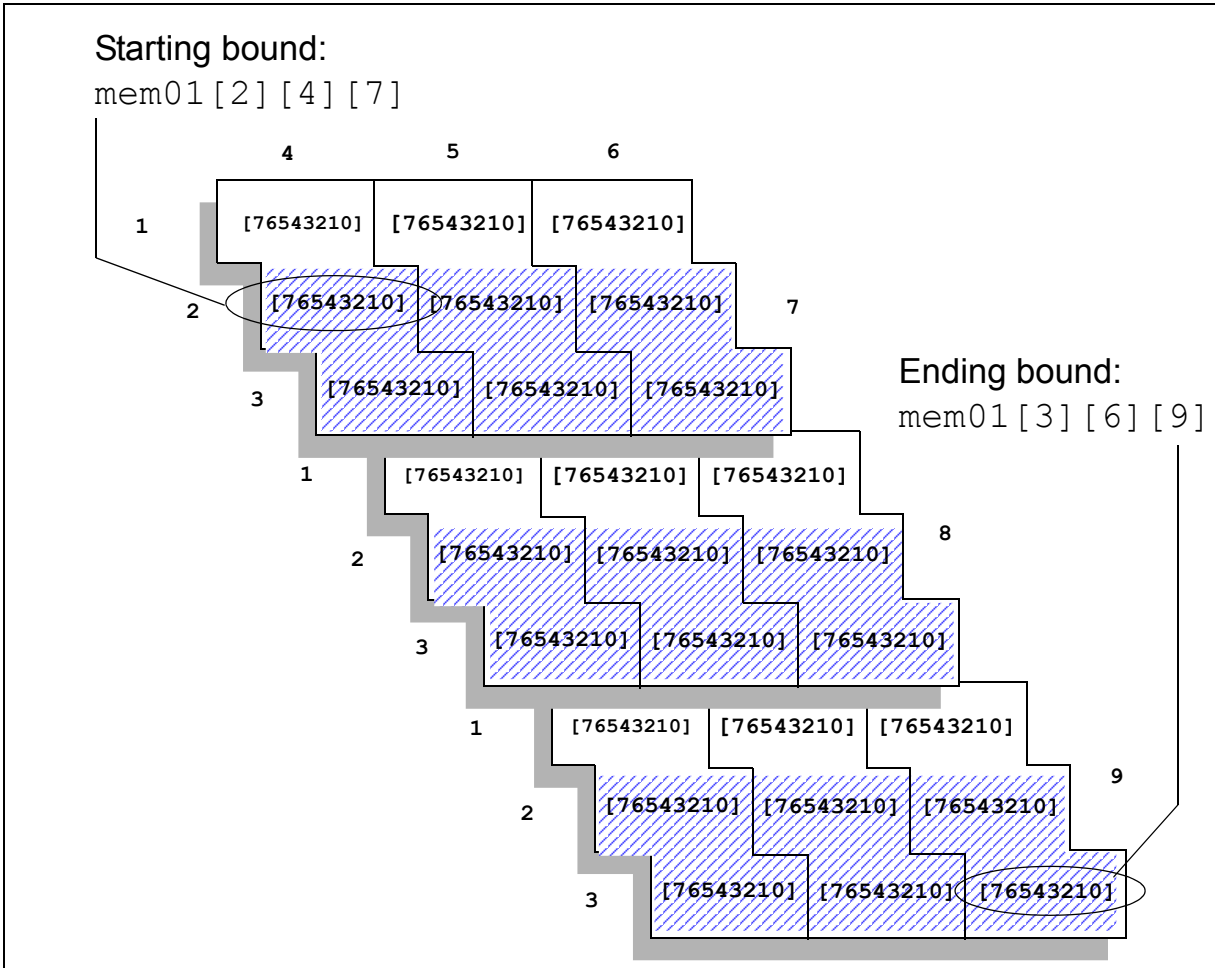


Example 6-3 `$vcdplusemon(mem01, addr1L, addr1R)`

```
addr1L = 2;
addr1R = 3;
$vcdplusemon( mem01, addr1L, addr1R );
// Records elements mem01[2][4][7] through mem01[3][6][9]
```

The elements highlighted by the  in the diagram in Figure 6-3 demonstrate Example 6-3.

Figure 6-3 `$vcdplusemon(mem01, addr1L, addr1R);`



Example 6-4 `$vcdplusemon(mem01, addr1L, addr1R, addr2L);`

```

addr1L = 2;
addr1R = 2;
addr2L = 5;
$vcdplusemon( mem01, addr1L, addr1R, addr2L );
// Records elements mem01[2][5][7] through mem01[2][5][9]

```


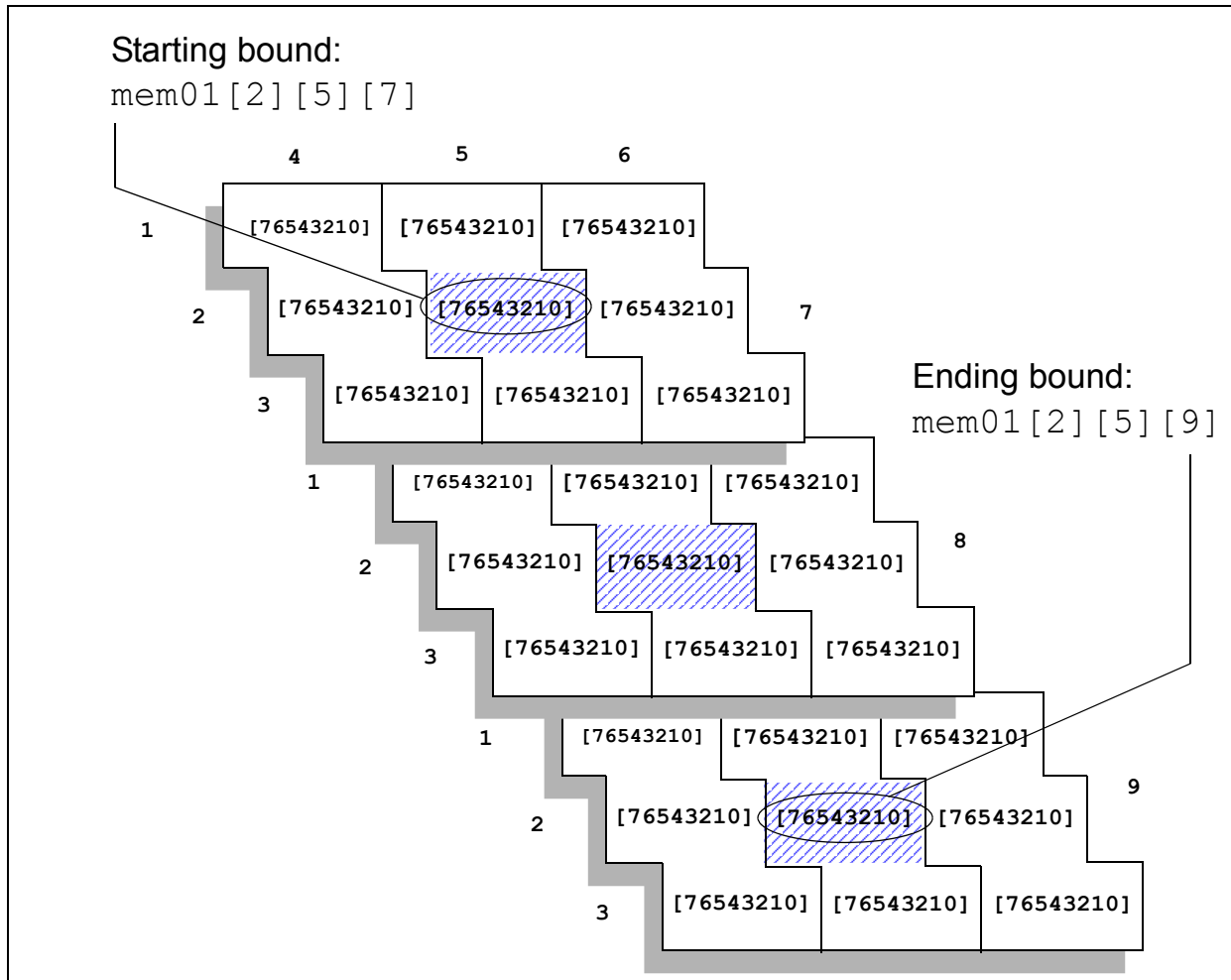
The elements highlighted by the  in the diagram in Figure 6-4 demonstrate Example 6-4.

Figure 6-4 `$vcdplusemon(mem01, addr1L, addr1R, addr2L);`



Example 6-5 `$vcdplusmemon(mem01, addr1L, addr1R, addr2L, addr2R)`

```
addr1L = 2;  
addr1R = 2;  
addr2L = 5;  
addr2R = 6;  
$vcdplusmemon( mem01, addr1L, addr1R, addr2L, addr2R );  
// Records elements mem01[2][5][7] through mem01[2][6][9]
```


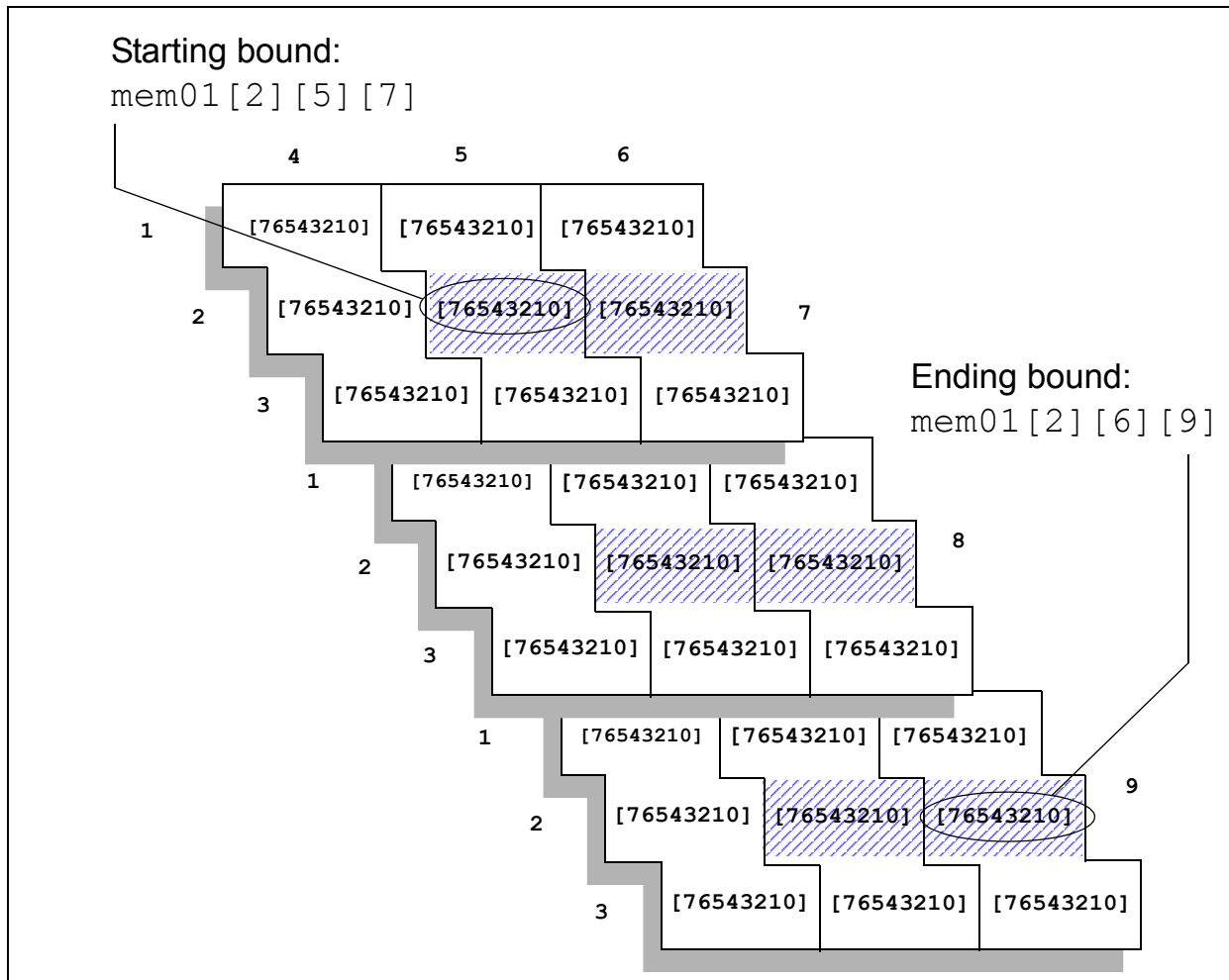
The elements highlighted by the  in the diagram in Figure 6-5 demonstrate Example 6-5.

Figure 6-5 `$vcdplusemon(mem01, addr1L, addr1R, addr2L, addr2R);`



Example 6-6 Selected element: `mem01[2][5][8]`

```

addr1L = 2;
addr1R = 2;
addr2L = 5;
addr2R = 5;
addr3L = 8;
addr3R = 8;
$vcdplusemon( mem01, addr1L, addr1R, addr2L, addr2R, addr3L
);
$vcdplusemon( mem01, addr1L, addr1R, addr2L, addr2R,

```

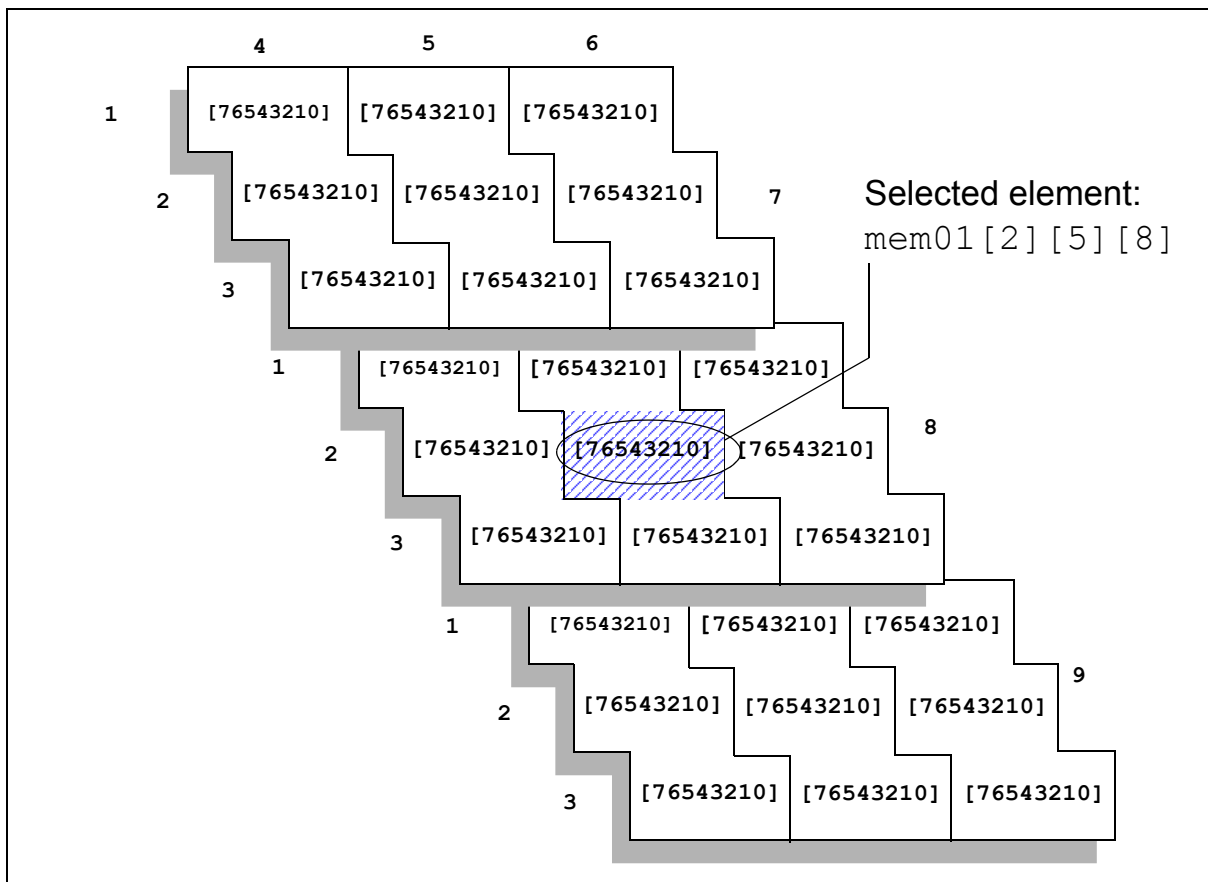
```

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

```

The element highlighted by the  in the diagram in Figure 6-6 demonstrates Example 6-6.

Figure 6-6 Selected element: mem01[2][5][8]



Using \$vcdplusmemorydump

The `$vcdplusmemorydump` 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 only once the complete set of multi-dimensional array elements to be dumped. You can specify multiple element subsets of an array using multiple `$vcdplusememorydump` commands, but they must occur in the same simulation time. In subsequent simulation times, `$vcdplusememorydump` 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.

Within VirSim, multi-dimensional arrays can be expanded in each dimension in much the same way as memories. By default, only the portions of the multi-dimensional array that have data are shown in VirSim.

System Tasks for Capturing Source Statement Execution Data

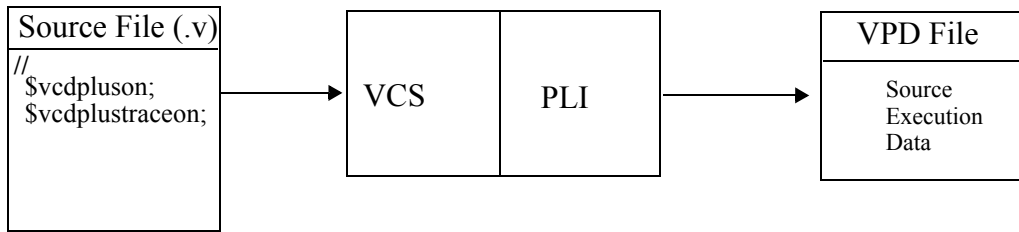
The Source Window requires the use of VPD options to save source hierarchy information and VPD tasks to capture source statement execution information in VPD files. Capturing source statement execution allows you to view and trace statement execution in the Source Window.

Note that saving statement execution data can significantly increase simulation time and VPD file size.

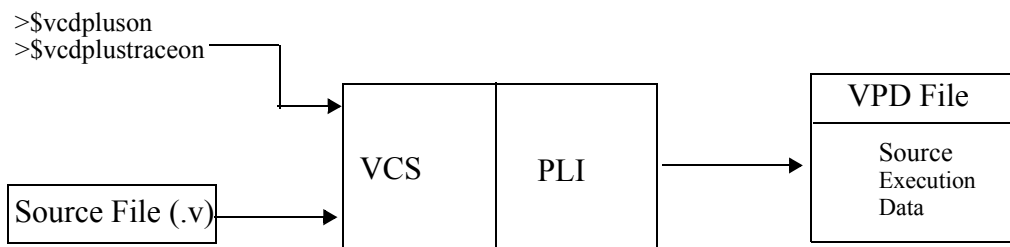
Capturing Source Statement Execution

The three ways to capture source statement execution are:

1. For viewing in post simulation mode, include `$vcdplustraceon`.



2. For viewing in post simulation mode, enter the appropriate trace task at the VCS command line.



3. For viewing in interactive mode in the DVE Source Window, Capture Line Data must be enabled (it is enabled by default). To also generate a VPD file for viewing in post simulation mode, include the appropriate trace task in the source file.

Source Statement System Tasks

Note:

For VCS you must supply the `-line` option when creating the simulation executable.

\$vcdplustraceon

The `$vcdplustraceon` task turns on line tracing. The VPD file saves line trace information.

Syntax:

```
$vcdplustraceon (<level>,<scope>*) ;
```

Here:

level

The number of hierarchy scope levels to descend to record line tracing (a zero value records all line tracing to the end of the hierarchy; default is 1 level).

scope

The name of the scope in which to record line tracing (default is 1 level).

Note:

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

\$vcdplustraceoff

The `$vcdplustraceoff` task turns off line tracing.

Syntax:

```
$vcdplustraceoff (<level>,<scope>*) ;
```

Here:

level

The number of hierarchy scope levels to descend to stop recording line tracing (a zero value stops the recording of all line tracing to the end of the hierarchy; default is 1 level).

System Tasks for Capturing Delta Cycle Information

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

`$vcdplusdeltacycleon`

The `$vcdplusdeltacycleon` task enables reporting of delta cycle information from the VCS CLI or the Verilog source code. It must be followed by the appropriate `$vcdpluson/$vcdplusoff` tasks.

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

Syntax:

```
$vcdplusdeltacycleon;
```

Note:

Delta cycle collection can start only at the beginning of a time sample. The `$vcdplusdeltacycleon` task must precede the `$vcdpluson` 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;
```

System Tasks for Capturing Unique Event Information

You can use the following VPD system tasks to capture unique events and glitch information.

`$vcdplusglitchon`

The `$vcdplusglitchon` task turns on checking for zero delay glitches and other cases of multiple transitions for a signal in one sample time. Glitch detection is automatically turned on when VCS executes `$vcdplusdeltacycleon` unless you have previously used `$vcdplusglitchon/off`. Once you use `$vcdplusglitchon/off`, DVE allows you explicit control of glitch detection.

When VCS detects a glitch for a signal, it records a zero delay glitch event. The default setting is not to perform zero delay glitch detection.

Syntax:

```
$vcdplusglitchon;
```

\$vcdplusglitchoff

The `$vcdplusglitchoff` task turns off checking for zero delay glitches. Glitch detection is automatically turned off when VCS executes `$vcdplusdeltacycleoff` unless you have previously used `$vcdplusglitchon/off`. Once you use `$vcdplusglitchon/off`, DVE allows you explicit control of glitch detection.

Syntax:

```
$vcdplusglitchoff;
```

\$vcdplusevent

The `$vcdplusevent` task allows you to record a unique event for a signal at the current simulation time unit. These events can be displayed in the Wave Window, Logic Browser, and Register Window.

There can be a maximum of 244 unique events, plus the predefined "glitch" event, which the PLI automatically generates, and a "Too many events" event, which is the default name for all unique events beyond the allowed 244.

Syntax:

```
$vcdplusevent (<signal>, "<event_name>",  
               "<severity><shape>");
```

Here:

signal

Any valid signal name.

event_name

A unique string which describes the event. This event name appears in the status bar of the Wave Window, Logic Browser, or Register Window when the mouse is placed on the event marker.

severity

A single character with legal values `E`, `W`, or `I`, which indicates the severity of the event. The severity of the event may be Error, Warning, or Information respectively. Colors associated with the severity level are set in the X Resource file. The defaults are Red=Error, Yellow=Warning, and Green=Information. If the severity is not interpretable, it defaults to `E`.

shape

A single character with legal values `S`, `T`, or `D` which indicates the geometry of the event as drawn by VirSim, and are Square, Triangle, and Diamond respectively. If the geometry is not interpretable, it will default to `T`.

Runtime Options

You can use specific command line options to generate VPD files. These options allow you to set the RAM buffer size, provide the VPD default file name, specify the VPD file size, ignore file calls, check licenses, and control what information is stored.

+vpdbufsize to Control RAM Buffer Size

To gain efficiency, VPD uses an internal buffer to store value changes before saving them on disk. The `+vpdbufsize` command modifies the size of the internal buffer.

Syntax:

+vpdbufsize*nn*

Here *nn* is buffer size in megabytes. The minimum size is the size required to store two value changes per signal and 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 to Set the Output File Name

The `+vpdfile` command allows you to specify the output file name.

Syntax:

+vpdfile*filename*

Here *filename* is the VPD filename (default is `vcdplus.vpd`). You must include the full file name with the `.vpd` extension.

+vpdfilesize to Control Maximum File Size

The `+vpdfilesize` command creates a VPD file, which never exceeds a specified file size *nn* megabytes. When the 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 can range from a few megabytes to a few hundred megabytes. Many DVE users can share the same VPD history file, which may be a reason for saving all time value changes when you simulate a design. You can save one history file for the design and overwrite it on each subsequent run.

Syntax:

```
+vpdfilesizesnn
```

Here *nn* is the file size in megabytes.

+vpdignore to Ignore \$vcdplus Calls in Code

The `+vpdignore` command instructs VCS to ignore any `$vcdplusxx` calls and license checking. By default, VCS checks out a VPD PLI license if there is a `$vcdplusxx` task in the Verilog source. In some cases, this statement is never executed and VPD PLI license checkout should be suppressed. The `+vpdignore` command performs the license suppression.

Syntax:

```
+vpdignore
```

+vpddrivers to Store Driver Information

By default, VPD records value changes only for the resolved value for each net. To also report value changes for all the drivers when there is more than one driver, use the `+vpddrivers` option during simulation. The driver values, for example, enable the Logic Browser to identify which drivers produce an undesired X on the resolved net.

This option affects performance and memory usage for larger designs or longer runs.

Syntax:

+vpddrivers

+vpdnoports to Eliminate Storing Port Information

By default, VPD stores the port type for each signal. When you use this option, the Hierarchy Browser views all signals as internal and not connected to a port.

The `+vpdnoports` option causes VPD to eliminate storing port information, which is used by the Hierarchy Browser to show whether a signal is a port and if so its direction. This option to some extent reduces simulation initialization time and memory usage for larger designs.

Syntax:

+vpdnoports

+vpdnocompress to Bypass Data Compression

By default, VPD compresses data as it is written to the VPD file. You can disable this feature by supplying the `+vpdnocompress` command line option.

Syntax:

+vpdnocompress

+vpdnostrengths to Not Store Strength Information

By default, VPD stores strength information on value changes to the VPD file. You can disable this feature by supplying the `+vpdnostrengths` command line option. Use of this option may lead to slight improvements in VCS performance.

Syntax:

`+vpdnostrengths`

VPD Methodology

The following information explains how to manage the DVE and VPD functions and how to optimize simulation and analysis.

Advantages of Separating Simulation From Analysis

Traditionally, interactive debugging has required a user to occupy one VCS license while simulating, thinking, resimulating, thinking...

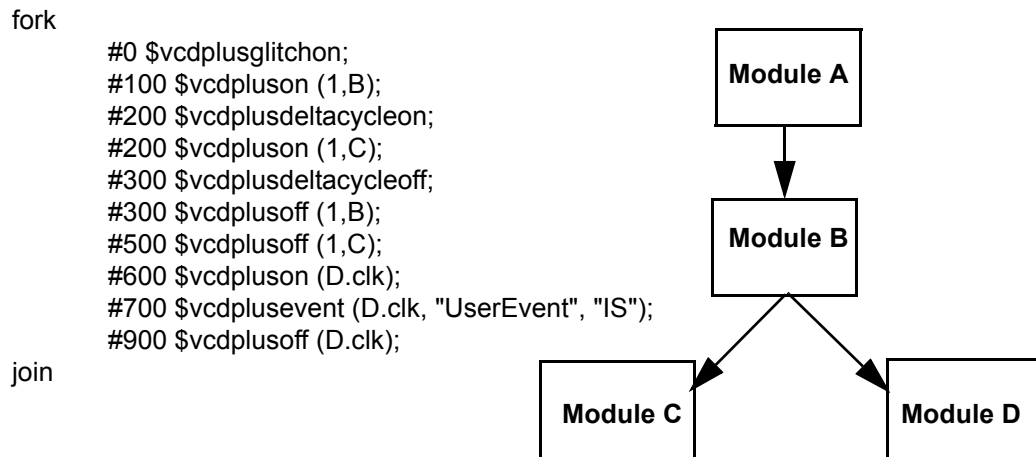
Simulating once and efficiently storing as much data as possible allows for a more efficient debug methodology that has the following features:

- VCS is used once and then released to other users.
- The analysis tool can go both forwards and backwards in time and analyze the complete set of data.
- The same set of data can be used by multiple users to debug one or more problems in parallel.

Conceptual Example of Using VPD System Tasks

The example in Figure 6-7, shows the entry of the `$vcdplus` system tasks in Module B scope. The dump saves all the variables in Module B from time 100 to 300, all variables in module C from time 200 to 500, and a single variable in module D1.clk from time 600 to 900. Zero delay glitch detection is on while value change data is recorded throughout the simulation. The VPD file stores delta cycle information starting at the first value change that occurs after time 200 and ending after the last value change during time 300. At time 700 a unique event is added to signal `D.clk`.

Figure 6-7 Example Definition of VPD Signal Capture (Recording)



Methods

You can implement signal data capture (recording) control in the source code and at the shell command line, as shown in the following examples.

- **Create a task in source:**

```
task sigon_instreg;
  begin
    $vcdpluson(test.risc1.instreg);
  end
endtask
```

Then call the task from source code.

```
initial
  sigon_instreg;
```

Or, enter the task name at the DVE interactive prompt.

```
C1> sigon_instreg;
```

- **Use a shell command argument to enable task execution:**

```
vcs -f run.f +signal_on
```

```
initial
  if ($test$plusargs("signal_on"))
    sigon_instreg;
task sigon_instreg;
  begin
    $vcdpluson(test.risc1.instreg);
  end
endtask
```

VPD On/Off PLI Rules

Follow these basic rules while using VPD On/Off PLI system tasks:

- You can insert the `$vcdpluson` and `$vcdplusoff` tasks in source code or enter it at the DVE interactive prompt.
- The `$vcdpluson` and `$vcdplusoff` tasks accept one level but multiple scopes/signals as arguments.
- The `$vcdpluson` and `$vcdplusoff` tasks, when applied to the same signals, toggle the recording on and off. The count for each signal is accumulative; (+,-). `..on/..on/..off` leaves the signal recording on. For example using the hierarchy of Figure 6-7, the following command sequence will still report on Module D since it is added twice but only removed once.

```
$vcdpluson(A); $vcdpluson(B); $vcdplusoff(D);
```

- On large designs, you should selectively turn signal data capture (dumping) on or off. Multiple use of `$vcdpluson` and `$vcdplusoff` allow on and off selection.
- The `$vcdpluson` and `$vcdplusoff` tasks executed in the same simulation time period may execute in any order. To ensure that one or the other executes last, separate them by at least one simulation time unit.
- Signals that are turned off may have signal value changes recorded if a higher/ lower level of the same signal is turned on.

Performance Tips

The following tips explain how to manage performance of VCS and VPD:

- Normally you should save data for all signals that you may require for analysis. The time range should be such that it very likely contains the origin of the problem.
- Generally, the bigger you make the RAM buffer size (via the `+vpdbufsize` option), the faster the simulation completes its run. The effects are so dependent on circuit and activity that rules-of-thumb do not apply. Synopsys suggests doubling the RAM size for the same simulation on your own design while measuring VCS performance to get a figure for an appropriate setting. When making this measurement, compare sizeable simulation runs to overcome the effects of compile time. Naturally, the above requires that you have physical memory to accommodate the simulation. Swapping significantly reduces performance.

Making the buffer size too large can cause excessive swapping, which can dramatically slow the simulation.

- Saving line-execution data enables more efficient debug of behavioral code. Access to such data allows breakpointing on particular activities in the code and stepping through the exact execution of the source. Correctly used, the cost of reporting on line execution data (slower simulation) more than pays for itself by enabling much faster location of the code deficiencies.

- Saving statement execution for an entire design can increase simulation time by eight times or more. To limit performance degradation, limit the use of statement saves to certain scopes. Instead of saving statement execution from time 0, turn on tracing just prior to the time of a suspected problem and off after that time.
- The file size increases from 200 to 500 percent when saving line execution data.
- Glitch detection and delta cycle may significantly increase the size of the VPD file.
- The `+vpdports` option costs some CPU time and memory in the initialization. It allows telling ports from internal signals in the hierarchy browser.
- The `+vpddrivers` option costs some CPU time and memory during the simulation. However, it allows visibility to the individual values of drivers of a multiply driven net.
- You can use the `+vpdfilesizesize` option (file wrap) in a verification environment where stimuli are automatically generated or read and the results are verified by the testbench. Then, you can stop the simulation when an error occurs, and the required history data will remain even in the relatively small file that is left.

EVCD File Generation

You can create an EVCD file for the entire design in the following ways:

- Using the runtime option `-dump_evcd`
- Using system tasks

Using the runtime option `-dump_evcd`

`-dump_evcd` writes an EVCD file for the instance/s specified as arguments to this option. You can specify more than one instance separated by “:” as shown below:

```
% executable -dump_evcd /top/dev1/intr1:/top/dev1/intr2
```

The above example dumps the port information of the modules `intr1` and `intr2`.

Use the compile time option `-enableEvcd`, to use `-dump_evcd` during runtime.

VCS, by default generates the EVCD file in the current working directory as “output.evcd” suffixed with the process id. However, you can overwrite this default file name using the runtime option `-dump_evcd_output filename`.

Using System Tasks

You can use `$dumpports` or `$lsi_dumpports` system tasks to generate EVCD files. Using system tasks you can generate multiple EVCD files for various module instances of the design. See [Appendix D, "Compiler Directives and System Tasks"](#).

7

VCD and VPD File Utilities

VCS comes with a number of utilities for processing VCD and VPD files. You can use these utilities to perform tasks like creating alternative VCD files, comparing the simulation data in two VCD, EVCD, or VPD files, easily viewing the data in a VCD file, and generating a VCD, EVCD, or VPD file that contains a selected subset of value changes found in a given input VCD, EVCD, or VPD file.

Note:

All the utilities are available in `$VCS_HOME/bin`.

This chapter 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]);
```

Here `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 vcdpost 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
```

Here 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 the ones 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
```

Here:

+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 uniquifies the identifier codes.

+unique

Specifies uniquifying the identifier codes. When you include this option without the `+scalar` option, `vcdpost` uniquifies 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:

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 vcdiff 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] [-ignoreregs] [-ingoreals]
[-ignorefunctaskvars] [-ignoretiming units] [-ignorestrength]
[-geninclude [filename]] [-spikes]
```

Options for Specifying Scope/Signal Hierarchy

The following options control how the 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 Dified

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 dified from the top of the design to the indicated scope. Note, all child scopes/signals of the indicated scope will be dified 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.

-prestroke

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 via `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.

-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.

-ignoretiming *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.

-ignorereggs

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 directionality 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 left-most 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 7-17](#).

The following is an example of the output from the vcat utility:

```
vcat expl.vcd

expl.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 you see, for example, 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 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 `vcad` to generate these files by doing the following:

1. Writing a configuration file.
2. Running `vcad` with the `-vgen` command line option.

Writing the Configuration File

The configuration file is named `vgen.cfg` by default and `vcad` 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 but if you do 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 `vcad` 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:

```
vcad 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` do 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;
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 any CLI or SCL dump commands to a VPD file.

The vcd2vpd Utility Syntax

```
vcd2vpd [-bmin_buffer_size] [-fmax_output_filesize] [-h]
[-m] [-q] [+delatcycle] [+glitchon] [+nocompress]
[+nocurrentvalue] [+bitrangenospace] [+vpdnoreadopt]
[+dut+dut_sufix] [+tf+tf_sufix] vcd_file vpd_file
```

Here:

-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_suffix*

Modifies the string identifier for the Device Under Test (DUT) half of the split signal. Default is "DUT".

+tf*+tf_suffix*

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, else 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 \$vcdpluson or any CLI or SCL dump commands to a VCD or EVCD file.

The vcd2vpd Utility Syntax

```
vcd2vpd [-h] [-q] [-s] [-x] [-xlrn] [+zerodelayglitchfilter]
[+morevhdl] [+start+value] [+end+value]
[+dumpsports+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 upper case VHDL objects to lower case.

+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 nonstandard 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. The syntax for this file is explained in the following section.

The Command file Syntax

Using a command file, you can:

- generate a VCD file for the whole design or for the specified instance/s.
- generate only the port information for the specified instance/s.
- generate an EVCD file for the specified instance/s.

Before writing a command file, please note the following:

- All commands must start as a first word in the line, and the arguments for these commands should be written in the same line.

Example:

```
dumpvars 1 adder4
```

- All comments must start with "//".

Example:

```
//Add your comment here  
dumpvars 1 adder4
```

- All comments written after a command, must be preceded by a space.

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` command 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` command 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` command 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

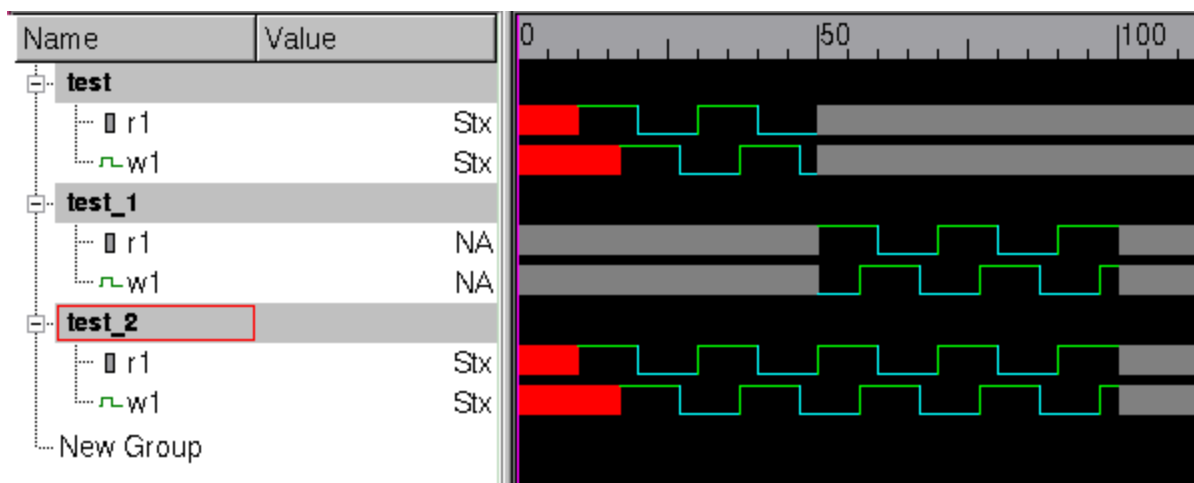
- `dumpports` is mutually exclusive with either the `dumpvars` or `dumpvcdports` commands. The reason is `dumpports` generates an EVCD file while both `dumpvars` and `dumpvcdports` generates standard VCD files.
- Escaped identifiers must include the trailing space.
- Any error parsing the file will cause the translation to terminate.

The vpdmerge Utility

Using `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 7-1, there are three signal groups for the same signals in different VPD files.

Figure 7-1 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 for the `vpdmerge` command line is as follows:

```
vpdmerge [-h] [-q] [-hier] [-v] -o merged_VPD_filename  
input_VPD_filename input_VPD_filename ...
```

Here:

-h

Displays a list of the valid options and their purpose.

-o *merged_VPD_filename*

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

There are the following restrictions for the vpdmerge utility:

- 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 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

Unified Command-Line Interface (UCLI)

The Unified Command-Line Interface (UCLI) enables consistent interactive simulation and post-processing using the UCLI interactive command language common to the following Synopsys verification technologies:

- VCS
- SystemVerilog
- NTB (OpenVera Language)
- DVE (Debug GUI)

UCLI is compatible with Tcl 8.3.

This chapter covers the following topics:

- [Compilation and Simulation Options for UCLI](#)
- [Using UCLI](#)

- [UCLI Interactive Commands](#)
- [UCLI Command-Alias File](#)
- [Operating System Commands](#)

Compilation and Simulation Options for UCLI

The VCS compilation and simulation options for UCLI are:

Compilation

-debug

Enables UCLI for interactive or post processing simulations. This option does not enable line stepping and setting of line breakpoints.

-debug_all

Enables UCLI for interactive simulations including line stepping, setting time, position, or trigger-type breakpoints, and the full debug mode.

Simulation

-ucli

Invokes UCLI for interactive simulations or debugging.

-l *logFilename*

Captures simulation output, and UCLI interactive commands used during simulation and responses to the commands.

-i *inputFilename*

Reads UCLI interactive commands from an interactive command file.

-k *keyFilename*

Records interactive commands used during a simulation to the file named *KeyFilename*, which can be used as the interactive command file in subsequent simulations.

Using UCLI

You can invoke UCLI as follows:

1. Compile a design with `vcs` using the `-debug` option.
2. Start the simulation with `simv` using the `-ucli` option.

To compile and simulate a design so that it results in an interactive simulation using UCLI, do the following:

Compilation

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

Simulation

```
% simv [simv_options] -ucli
```

UCLI Interactive Commands

The following is a list of the UCLI interactive commands:

Tool Invocation Commands

start *exe_name* [*options*]

Invokes the simulation executable *exe_name* with the specified options.

restart [*options*]

Restarts simulations.

Session Management Commands

save [*file_name*]

Saves simulation state in file *file_name*.

restore [*file_name*]

Restores a simulation state saved in file *file_name*.

Note:

Simulations run in UCLI mode can only be saved and restored from the UCLI prompt, not using \$save/\$restart in Verilog, and cannot be restarted by running the simulator with "-r savefile", nor by executing the save file.

Tool Advancing Commands

step

Advances the simulation by one line.

next

For Verilog designs, *next* has the same functionality as *step*.

run [-relative | -absolute time] [-posedge |
-negedge | -change] [*path_name*]
Advances simulation to a point specified by time or edge of signal
path_name.

finish
Terminates a simulation, but remains in Tcl debug environment.

Navigation Commands

scope [-up [*level*] | active] [*path_name*]
Shows or sets current scope to instance *path_name*.

thread [*thread_id*][-active][-attach *thread_id*]
Displays thread with ID *thread_id* or all threads when no ID is
provided.

Signal/Variable/Expression Commands

get *path_name* [-radix *radix*]
Returns current value of signal/variable/net/reg *path_name*.

change [*path_name value*]
Deposits value on signal/variable/net/reg *path_name*.

force *path_name value* [time { , value time }*
[-repeat delay]][-cancel time][-deposit]
[-freeze]
Forces signal/variable/net/reg *path_name* with value *value*.

release [*path_name*]
Releases signal/variable/net/reg *path_name* from the value
assigned using force command.

sexpr [-radix] [*expression*]
Displays the result in base radix of expression *expression*.

call [*\$cmd(...)*]
Calls a Verilog task.

Tool Environment Array Commands

senv [*element*]
Displays the environment array element.

Breakpoint Commands

stop [**-file** *file_name*] [**-line** *num*] [**-instance** *path_name*] [**-thread** *thread_id*] [**-condition** *expression*]
Sets and displays breakpoints based on file *file_name*, source code line with number *num*, instance *path_name*, thread with ID *thread_id* or condition expression.

Signal Value and Memory Dump Specification Commands

dump [**-file** *file_name*] **-add** [*list_of_path_names* **-fid** *fid* **-depth** *levels* | *object* **-aggregates** **-close**] [**-file** *file_name*] [**-autoflush** *on*] [**-file** *file_name*] [**-interval** *seconds*] [**-fid** *fid*]
Dumps values of signals/variables/nets/regs listed in *list_of_path_names* in file *file_name*.

memory [**-read**|**-write** *nid*] [**-file** *file_name*] [**-radix** *radix*] [**-start** *start_address*] [**-end** *end_address*]
Loads or writes memory from or to file *file_name*, respectively, between locations *start_address* and *end_address*.

Design Query Commands

show [**-options**] *path_name*
Displays value of object *path_name*. Also provides full static information of instances and objects.

drivers *path_name* [-full]
Displays drivers of object *path_name*.

loads *path_name* [-full]
Displays load on object *path_name*.

Macro Control Routines

do [-trace|-traceall] *file_name* [macro
parameters] [-trace|-traceall [on|off]]
Reads macro file *file_name*.

onbreak [*commands*]
Executes one or more commands when a breakpoint, \$stop task or Ctrl-c is encountered while executing a macro file.

onerror [*commands*]
Executes one or more commands when an error is encountered while executing a macro file.

pause
Interrupts execution of a macro file.

resume
Resumes execution of a macro file after a breakpoint, error or pause.

abort [*n* | all]
Stops execution of a macro file and discards remaining commands in it.

status [*file* | *line*]
Displays stack of nested macro files, along with file names and position of interruption in each.

Helper Routine Commands

help *-[full command]*

Displays information on all commands or specified command.

alias *[UCLI_command]*

Creates an alias *alias* for command *UCLI command*.

config

Displays current settings of all variables.

Specman Interface Command

sn

Switches to Specman prompt.

UCLI Command-Alias File

You can call UCLI commands with aliases defined in a command-alias file. You can either create this file or use the default command-alias file.

Default Alias File

The default alias file `.uclirc` in the VCS installation directory contains default aliases for UCLI commands. You can edit this file to add custom aliases for UCLI commands. By default, UCLI looks for the alias file at the following three places in the given order:

- Current work directory
- User's home directory
- VCS installation directory

Operating System Commands

Operating System (OS) commands can be used at the UCLI prompt as follows:

```
ucli% exec OS_command
```

In the interactive mode, OS commands run automatically. To disable automatic execution of OS commands, set the `auto_noexec` variable as follows:

```
set ::auto_noexec anything
```


9

Using the Old Command Line Interface (CLI)

VCS provides the non-graphical debugger or CLI (Command Line Interface) for debugging your design. This chapter covers the following topics:

- [CLI Commands](#)
- [Command Files](#)
- [Key Files](#)
- [Debugging a Testbench Using the CLI](#)

Note:

There now is a Unified Command Line Interface for debugging commands. It is unified in that the same command line interface works for VCS, VCS MX, and Vera. It has more commands than the CLI. See [Chapter 8, "Unified Command-Line Interface \(UCLI\)"](#).

CLI Commands

You can use basic Command Language Interface (CLI) commands to perform the following tasks:

- Navigate the design and display design information
- Show and retrieve simulation information
- Set, display and delete breakpoints
- Display object data members
- Set and print values of variables
- Traverse call-stacks
- Show and terminate threads
- Access events

Navigating the Design and Displaying Design Information

help

Displays the list of all commands and their meanings.

info

Displays time and scope information.

For example:

```
cli> info
Current time is 100000
Current thread is #3
Current scope is memsys_test_top
```

line

Toggles line tracking.

For example:

```
cli> line
Line tracking is now ON.
cli> line
Line tracking is now OFF.
```

list [-*n* | *n*]

Lists 10 lines starting with the current line.

-n

Lists *n* lines above the current position.

n

Lists *n* lines starting with the current line.

print [%**b|c|t|f|e|g|d|h|x|m|o|s|v**] *net_or_reg*

Shows the current value of net or register in the specified format.

next

Next line.

step [-**thread** *thread-id* | **up**]

When entered without a qualifier, moves through all traceable lines according to the order of event execution.

-thread *thread-id*

Steps in the specified thread while skipping statements in other threads.

For example:

```
cli_15 > step -thread
mem_add1[10011011] is 10011011
[memsys0.vr:91]
```

up

Steps out of current automatic task or function.

For example:

```
cli_65 > step up
Time break at time 100200  breakpoint #1 break #50
##100250
[cpu.vr:79]
```

Showing and Retrieving Simulation Information

show [drivers|loads|ports|scopes|variables|break|?]

drivers *net_or_reg*

Shows the value and strength of the net or register. For nets it also shows the line number in the source code of the statement that is the source of the value that propagated to this net.

For example:

```
cli_24 > show drivers ramData
ramData[7] (ram_test_top.dut.u3) = StX
StX <- (ram_test_top.dut.u3) sram.v: 10 (ASSIGN)
ramData[6] (ram_test_top.dut.u3) = StX
StX <- (ram_test_top.dut.u3) sram.v: 10 (ASSIGN)
ramData[5] (ram_test_top.dut.u3) = StX
StX <- (ram_test_top.dut.u3) sram.v: 10 (ASSIGN)
ramData[4] (ram_test_top.dut.u3) = StX
StX <- (ram_test_top.dut.u3) sram.v: 10 (ASSIGN)
ramData[3] (ram_test_top.dut.u3) = StX
StX <- (ram_test_top.dut.u3) sram.v: 10 (ASSIGN)
ramData[2] (ram_test_top.dut.u3) = StX
StX <- (ram_test_top.dut.u3) sram.v: 10 (ASSIGN)
ramData[1] (ram_test_top.dut.u3) = StX
StX <- (ram_test_top.dut.u3) sram.v: 10 (ASSIGN)
ramData[0] (ram_test_top.dut.u3) = StX
StX <- (ram_test_top.dut.u3) sram.v: 10 (ASSIGN)
```


loads *nid*

Displays the loads for the specified signal.

For example:

```
cli_23>show loads ramData
ramData[7] (ram_test_top.dut.u3) = StX
ramData[6] (ram_test_top.dut.u3) = StX
ramData[5] (ram_test_top.dut.u3) = StX
ramData[4] (ram_test_top.dut.u3) = StX
ramData[3] (ram_test_top.dut.u3) = StX
ramData[2] (ram_test_top.dut.u3) = StX
ramData[1] (ram_test_top.dut.u3) = StX
ramData[0] (ram_test_top.dut.u3) = StX
```

ports

Shows the port identifiers of the instance, that is, the current scope and whether they are input, output, or inout ports, listed as IN, OUT, and INOUT.

scopes

Shows the module instances in the current scope by their module identifier and module instance identifier.

variables

Shows the nets and registers declared in the current scope.

break

Lists the hierarchical names of the nets and registers with a breakpoint and the breakpoint number for these breakpoints.

?

Displays this list of arguments to the `show` command and briefly describes what they do.

For example:

```
cli> show variables
Int          vtb_temp_int2
```

**show [allvariables|mailboxes|semaphores|threadsevent]
allvariables**

Shows nets and registers in the current module and its parents.

For example:

```
cli> show allvariables
listing variables for :
memsys_test_top.vshell.\cpu::release_bus
.unnamed$_29
      Int          vtb_temp_int2
      listing variables for :
memsys_test_top.vshell.\cpu::release_bus
      listing variables for : memsys_test_top.vshell
      Wire          SystemClock
      Wire          \memsys.adxStrb
      Wire [7:0]    \memsys.busAddr
      Wire [7:0]    \memsys.busData
      Wire          \memsys.busRdWr_N
      Wire          \memsys.clk
      Wire [1:0]    \memsys.grant
      Wire [1:0]    \memsys.request
      Wire          \memsys.reset
      Class         arb0
      Class         arb1
      Class         cpu0
      Class         cpu1
      listing variables for : memsys_test_top
      Reg           SystemClock
      Wire          adxStrb
      Wire [7:0]    busAddr
      Wire [7:0]    busData
      Wire          busRdWr_N
      Wire          clk
      Wire [1:0]    grant
      Wire [1:0]    request
      Wire          reset
```

mailboxes *[m_id]*

Displays information about all mailboxes or the identified mailbox.

semaphores *[n_id]*

Displays information about all semaphores or the identified semaphore.

For example:

```
cli_33 > show semaphore
semaphore id = 1  keys available 0
blocked threads:
thread #2:      memsys0.vr: 65
```

thread *[t_id]*

Displays all threads or the identified thread and the status.

For example:

```
cli_32 > show thread
thread #1 memsys_test_top.vshell.check_all
         [ready]      memsys0.vr: 99
thread #2 memsys_test_top.vshell.check_all
         [blocked]   memsys0.vr: 65
thread #3 memsys_test_top.vshell.\cpu::release_bus
         .unnamed$$_29 [current]      cpu.vr: 73
```

event *n_id*

Shows the changes in value for the identified event.

Setting, Displaying and Deleting Breakpoints

break # *relative time* | [**@posedge** | **@negedge**] *signal*

Sets a repeating breakpoint.

break -thread *thread_id*

Sets breakpoint in the specified thread.

break at *file:lineno/lineno -thread thread_id*

Sets breakpoint at the line number of the file and thread mentioned.

break at *filename:lineno*

Sets a breakpoint in the identified file at the specified line number.

break in *class:task/function*

Sets a breakpoint in class at the identified task or function.

For example:

```
cli_55 > break in check_all
set break #6 in check_all
```

break in *scope*

Sets a breakpoint in the specified scope.

break in *scope -thread thread_id*

Sets a breakpoint in the scope of the thread.

show break

Displays all the break points

delete *breakpoint_number | all*

Deletes the identified breakpoint or all breakpoints.

tbreak [*#relative_time* | *##absolute_time* | **@posedge** | **@negedge**] *net_or_reg*

Sets a one shot breakpoint. This command is identical to the `once` command.

Displaying Object Data Members

print this

In Vera code, prints the current object data members.

For example:

```
cli_141 > break in cpu::new
set break #1 in cpu::new
cli_142 > print this
this = {
  localarb: <Class Type bus_arb>
  cpu_id: 00000000
  address: 69
  data: 30
  delay: 00000003
}
```

Setting and Printing Values of Variables

set *variable = value*

Sets variable values.

print *variable*

Displays variable values.

Traversing Call-stacks

stack

Prints task/function call traceback.

For example:

```
cli_129 > stack
#0 in \cpu::release_bus at cpu.vr:73
```

```
#1 in check_all at memsys0.vr:68
#2 in memsys_test at memsys0.vr:19
#3 in memsys_test_top.vshell
```

upstack

Goes up the call stack.

For example:

```
cli_131 > upstack
#1 in check_all at memsys0.vr:68
```

downstack

Goes down the call stack.

For example:

```
cli_132 > downstack
#0 in \cpu::release_bus at cpu.vr:73
```

Showing and Terminating Threads

show thread [*thread_id*]

Prints information about the specified threads.

For example:

```
li_119 > show thread 2
thread #2
memsys_test_top.vshell.\cpu::release_bus
  [ready]                cpu.vr: 73
```

thread *thread_id*

Options context to the task specified. *thread_id* is blocked

terminate [*thread_id*]

Terminates the specified thread.

Accessing Events

`trigger(0|1|2|3|4, event variable)`

Triggers an event in the testbench according to the following:

0 -> Off

1 -> On

2 -> One shot

3 -> One blast

4 -> Handshake

Command Files

It is possible to create, in the working directory, a `.vcsrc` file containing CLI commands that VCS executes on entry to the CLI. This is useful for specifying alias commands to customize the command language. Any CLI command can appear in this file.

Within the CLI, use the `source` command at any time to read in a file that contains CLI commands. The `-i` runtime option is shorthand to specify a source file to be read upon entry to the CLI.

If a `.vcsrc` file exists in the working directory when a simulation is run, the executable reads it and executes commands at time zero before executing any commands in a `-i` file.

Example 9-1 Interactive Debugging Example

The following is an example of the use of the interactive debugger:

```
% more a.v
```

```

module top;
    reg a;
    reg [31:0] b;
    initial begin
        a = 0;
        b = 32'b0;
        #10
        if (a) b = 32'b0;
    end
endmodule
% vcs +cli+2 a.v

```

<<Details of VCS compilation omitted.>>

```

% simv -s
$stop at time 0
cli_0 > scope
Current scope is top
cli_1 > show var
Reg                a
Reg [31:0]         b
cli_2 > once #1
cli_3 > .
Time break at time 1 breakpoint #1 tbreak ##1
cli_4 > print a
a: 0
cli_5 > set a=1
cli_6 > print a
a: 1
cli_7 > tbreak b
cli_8 > .
Value break time 10 breakpoint #2 tbreak top.b
cli_9 > print b
b: 00000001
cli_10 > quit
$finish at simulation time 10
                V C S      S i m u l a t i o n  R e p o r t
Time: 10
CPU Time: 0.150 seconds; Data structure size: 0.0Mb

```

Key Files

When you enter CLI commands (or commands in the DVE Interactive window), VCS by default records these commands in the `vcs.key` file that it writes in the current directory.

The purpose of this file is to enable you to quickly enter all of the interactive commands from another simulation of your design by including the `-i` runtime option with this file as its argument.

You can use the `-k` runtime option to specify a different name or location for the `vcs.key` file. You can also use this option to tell VCS not to write this file. For details on using the `-i` and `-k` runtime options, see [Appendix C, "Simulation Options"](#).

Debugging a Testbench Using the CLI

The interactive non-graphical debugging command line interface (CLI) capability in VCS also covers testbench files. It is similar in concept to UNIX debuggers such as `dbx` and `gdb`. You can enter the CLI at runtime for debugging a testbench, provided you have enabled the CLI at compile time. The command language not only allows you to set breakpoints, examine the values of registers and wires, and change register values, but also enables you to examine testbench objects and data types. Since the CLI covers both the design and the testbench, you can cross over from the design into the testbench or vice-versa during the debug process.

Non-Graphical Debugging With the CLI

In order to use the CLI:

- Enable it at compile time with the options `+cli` and `-line`.
- Include the `-s` on the runtime command line (e.g. `simv -s`).

For example:

When compiling both the testbench and the design together, the command lines are:

```
% vcs -ntb +cli -line sram.v sram.test_top.v sram.vr
% simv -s
```

When compiling the testbench separately from the design, the command lines are:

```
% vcs -ntb_cmp +cli -line -ntb_sname sram_test sram.vr
% vcs -ntb_vl +cli -line sram.v sram.test_top.v sram.vshell
% simv -s +ntb_load=./libtb.so
```

Using the CLI, Example 1

Example 9-2 is a testbench, containing mailboxes and triggers. It is in the file `memsys1.vr`, which is a part of the Native Testbench tutorial example in your VCS installation under `$VCS_HOME/doc/examples/nativetestbench/tutorial`.

Example 9-2 Testbench Containing Mailboxes and Triggers

```
#define OUTPUT_SKEW #1
#define INPUT_SKEW #-1
#define INPUT_EDGE PSAMPLE
#include <vera_defines.vrh>

#include "memsys.if.vrh"
```

```

#include "port_bind.vr"
#include "cpu.vr"

program memsys_test
{ // Start of memsys_test

    cpu cpu0 = new (arb0, 0);
    cpu cpu1 = new (arb1, 1);

    init_ports();
    reset_sequence();
    check_all() ;

} // end of program memsys_test

// Don't allow inputs to dut to float
task init_ports () {
    printf("Task init_ports\n");
    @(posedge memsys.clk);
    memsys.request = 2'b00;
    memsys.busRdWr_N = 1'b1;
    memsys.adxStrb = 1'b0;
    memsys.reset = 1'b0;
}

task reset_sequence () {
    printf("Task reset_sequence\n");
    memsys.reset = 0;
    @1 memsys.reset = 1;
    @10 memsys.reset = 0;
    @1 memsys.grant == 2'b00; //check if grants are 0's
}

task check_all () {

    integer mboxId, randflag;
    event CPU1done;

    printf("Task check_all:\n");
    mboxId = alloc(MAILBOX, 0, 1);
}

```

```

fork
  { // fork process for CPU 0
    repeat(256) {
      randflag = cpu0.randomize();
      cpu0.request_bus();
      cpu0.writeOp();
      cpu0.release_bus();
      mailbox_put(mboxId, cpu0.address);
      mailbox_put(mboxId, cpu0.data);
      mailbox_put(mboxId, cpu0.delay);
      sync(ALL, CPU1done);
      trigger(OFF, CPU1done);
      cpu0.delay_cycle();
    }
  }

  { // fork process for CPU 1
    repeat(256) {
      mailbox_get(WAIT, mboxId, cpu1.address, CHECK);
      mailbox_get(WAIT, mboxId, cpu1.data, CHECK);
      mailbox_get(WAIT, mboxId, cpu1.delay, CHECK);
      cpu1.request_bus();
      cpu1.readOp();
      if (memsys.busData == cpu1.data)
        printf("\nThe read and write cycles finished
successfully\n\n");
      else
        printf("\nThe memory has been corrupted\n\n");
      cpu1.release_bus();
      trigger(ON, CPU1done);
      cpu1.delay_cycle();
    }
  }
  join
}

```

Use the following command line to compile the design and the testbench in Example 9-2:

```
%vcs -ntb +cli -line -f memsys.f memsys.test_top.v memsys1.vr
```

Use the following command line to run the simulation with debug:

```
% simv -s
```

The following is the output while using the CLI:

```
Chronologic VCS simulator copyright 1991-2003  
Contains Synopsys proprietary information.  
Compiler version 7.1_Beta2; Runtime version 7.1_Beta2; Oct  
16 16:55 2003
```

```
$stop at time 0  
cli_0 > break in memsys_test  
set break #1 in memsys_test
```

```
cli_1 > break in check_all  
set break #2 in check_all
```

```
cli_2 > break at memsys1.vr69:  
set break #3 at memsys1.vr:69
```

```
cli_3 > cont  
Constructing new CPU.  
WARNING: Ignoring seed value 0. Seeding with value 1. Seed  
must be greater than 0.  
Constructing new CPU.  
Scope break at time 0 breakpoint #1 break in  
memsys_test_top.vshell.memsys_test
```

```
cli_4 > .  
Task init_ports  
Task reset_sequence  
Scope break at time 1250 breakpoint #2 break in  
memsys_test_top.vshell.check_all
```

```
cli_5 > .  
Task check_all:  
CPU          0 requests bus on arb0  
CPU          0 writeOp: address 0b data df  
CPU0 is writing  
WRITE address = 00b, data = 0df
```

```
CPU          0 releases bus on arb0
Line break at time 2050 breakpoint #3 break at memsys1.vr:69
```

```
cli_6 >show mailboxes
mailbox id: 1  data available: 1
  data: -->1
  blocked threads: NONE
```

```
cli_7 > quit
$finish at simulation time          2050
          V C S   S i m u l a t i o n   R e p o r t
```

Using the CLI, Example 2

Example 9-3 is a testbench containing semaphores. It is in the file memsys0.vr, which is a part of the Native Testbench tutorial example in your VCS installation under `$VCS_HOME/doc/examples/nativetestbench/tutorial`.

Example 9-3 Testbench Containing Semaphores

```
#define OUTPUT_EDGE  PHOLD
#define OUTPUT_SKEW  #1
#define INPUT_SKEW   #-1
#define INPUT_EDGE   PSAMPLE
#include <vera_defines.vrh>

#include "memsys.if.vrh"
#include "port_bind.vr"
#include "cpu.vr"

program memsys_test
{  // Start of memsys_test

    cpu cpu0 = new (arb0, 0);
    cpu cpu1 = new (arb1, 1);

    init_ports();
    reset_sequence();
    check_all() ;
}
```

```

} // end of program memsys_test

// Don't allow inputs to dut to float
task init_ports () {
    printf("Task init_ports\n");
    @(posedge memsys.clk);
    memsys.request = 2'b00;
    memsys.busRdWr_N = 1'b1;
    memsys.adxStrb = 1'b0;
    memsys.reset = 1'b0;
}

task reset_sequence () {
    printf("Task reset_sequence\n");
    memsys.reset = 0;
    @1 memsys.reset = 1;
    @10 memsys.reset = 0;
    @1 memsys.grant == 2'b00; //check if grants are 0's
}

task check_all () {

    integer semaphoreId, randflag;
    event CPUldone;
    bit[7:0] mem_add0[], mem_add1[];

    printf("Task check_all:\n");
    semaphoreId = alloc(SEMAPHORE, 0, 1, 1);

    fork
        { // fork process for CPU 0
            repeat(256) {
                randflag = cpu0.randomize();
                printf("\n THE RAND MEM0 ADD IS %b \n\n", cpu0.address);
                if (mem_add0[cpu0.address] != cpu0.address)
                {
                    mem_add0[cpu0.address] = cpu0.address;
                    printf("\n mem_add0[%b] is %b \n\n", cpu0.address,
mem_add0[cpu0.address]);
                }
                else
                {

```



```

        cpul.delay_cycle();
    }
}
join
}

```

Use the following command line to compile the design and the testbench:

```
% vcs -ntb +cli -f memsys.f memsys.test_top.v memsys0.vr
```

Use the following command line to run the simulation with debug:

```
% simv -s
```

The following is the output while using the CLI:

```

Chronologic VCS simulator copyright 1991-2003
Contains Synopsys proprietary information.
Compiler version 7.1_Beta2; Runtime version 7.1_Beta2; Oct
16 17:09 2003

$stop at time 0
cli_0 > break in memsys_test
set break #1 in memsys_test

cli_1 > break in check_all
set break #2 in check_all

cli_2 > break at memsys0.vr:99
set break #3 at memsys0.vr:99

cli_3 > cont
Constructing new CPU.
WARNING: Ignoring seed value 0. Seeding with value 1. Seed
must be greater than 0.
Constructing new CPU.
Scope break at time 0 breakpoint #1 break in
memsys_test_top.vshell.memsys_test

cli_4 > .

```

```
Task init_ports
Task reset_sequence
Scope break at time 1250 breakpoint #2 break in
memsys_test_top.vshell.check_all
```

```
cli_5 >.
Task check_all:
```

```
THE RAND MEM0 ADD IS 00001011
```

```
mem_add0[00001011] is 00001011
```

```
CPU          0 requests bus on arb0
```

```
THE RAND MEM1 ADD IS 10010111
```

```
mem_add1[10010111] is 10010111
```

```
CPU          0 writeOp: address 0b data df
```

```
CPU0 is writing
```

```
WRITE address = 00b, data = 0df
```

```
CPU          0 releases bus on arb0
```

```
CPU          0 requests bus on arb0
```

```
CPU          0 readOp: address 0b data df
```

```
READ address = 00b, data = 0df
```

```
CPU          0 releases bus on arb0
```

```
CPU          0 Delay cycle value:          1
```

```
CPU          1 requests bus on arb1
```

```
delay =          1
```

```
THE RAND MEM0 ADD IS 10110001
```

```
mem_add0[10110001] is 10110001
```

```
CPU          1 writeOp: address 97 data f1
```

```
CPU1 is writing
```

```
WRITE address = 097, data = 0f1
```

```
CPU          1 releases bus on arb1
```

```
CPU          1 requests bus on arb1
```

```
CPU          1 readOp: address 97 data f1
READ address = 097, data = 0f1
CPU          1 releases bus on arb1
Line break at time 5050 breakpoint #3 break at memsys0.vr:99
```

```
cli_6 > show semaphores
semaphore id = 1 keys available 0
  blocked threads:
    thread #3:      memsys0.vr: 91
```

```
cli_7 > quit
$finish at simulation time          5850
      V C S   S i m u l a t i o n   R e p o r t
3
```

Using the Old Command Line Interface (CLI)

9-24

10

Post-Processing

- Use the `$vcdpluson` system task to generate VPD files. For more details on this system task, see "[System Tasks for VPD Files](#)" in [Appendix D](#).

If you enable it, the VPD file contains simulation results from the design. The `$vcdplustraceon` system task records the order in which the source code lines are executed. Therefore, you can see the order in post processing. You can enter these system tasks either in your Verilog source code or at the CLI prompt using a different syntax. The default name of the generated VPD file is `vcdplus.vpd`.

This chapter describes the simulation history file formats and how you generate them. It covers the following:

- [VPD](#)
- [eVCD](#)

- [Line Tracing](#)
- [Delta Cycle](#)
- [Verilog HDL offers the advantage of having the ability to access any internal signals from any other hierarchical block without having to route it through the user interface.](#)
- [You can generate an EVCD file using the `\$dumpports` or `\$lsi_dumpports` system tasks](#) See “[Verilog HDL offers the advantage of having the ability to access any internal signals from any other hierarchical block without having to route it through the user interface.](#)” on page 10-4.

VPD

You can create a single VPD file for the entire design using the `$vcdpluson` system task. To create a dump file, use the `$dumpvars` system task. To enable post-processing for the design, use the `-PP` option as a compile-time option.

Delta Dumping Supported in VPD Files

VCS supports delta dumping in VPD files for post-processing using the `$vcdplusdeltacycleon` system task.

The `$vcdplusevent` system task displays, in DVE, 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.

Syntax:

```
$vcdplusevent(net_or_reg,"event_name", "<E|W|I><S|T|D>");
```

eVCD

You can use `$dumpports` or `$lsi_dumpports` system tasks to generate EVCD files. Using system tasks you can generate multiple EVCD files for various module instances of the design.

Line Tracing

You can enable line tracing for the design using `-debug_all` or `-debug` as a compile-time option. The `-debug_all` option greatly degrades the simulation performance and therefore you should use it only for debugging. You can also use `-PP -line` as compile-time options, instead of `-debug_all`.

Delta Cycle

VCS supports delta dumping in VPD files for post-processing using the `$vcdplusedeltacyclone` system task. Like other system tasks, this task can either be used in the design or can be entered at the CLI prompt. However, to enable delta cycle dumping, `$vcdplusedeltacyclone` should be entered before entering the `$vcdpluson` system task.

Verilog HDL offers the advantage of having the ability to access any internal signals from any other hierarchical block without having to route it through the user interface.

11

Race Detection

VCS provides a dynamic race detection tool that finds race conditions during simulation and a static race detection tool that finds race conditions by analyzing source code during compilation. This chapter describes these two tools in the following sections:

- [The Dynamic Race Detection Tool](#)
- [The Static Race Detection Tool](#)

The Dynamic Race Detection Tool

The dynamic race detection tool finds two basic types of race conditions during simulation:

- **read - write race condition**
This occurs when a procedural assignment in one always or initial block, or a continuous assignment assigns a signal's value to another signal (read) at the same time that a procedural assignment in another always or initial block, or another continuous assignment assigns a new value to that signal (write). For example:

```
initial
#5 a = 0; // write operation to signal a
```

```
initial
#5 b = a; // read operation of signal a
```

In this example, at simulation time 5, there is both a read and a write operation on signal a. When simulation time 5 is over you do not know if signal b will have the value 0 or the previous value of signal a.

- **write - write race condition**
This occurs when a procedural assignment in one always or initial block, or a continuous assignment assigns a value to a signal (write) at the same time that a procedural assignment in another always or initial block, or another continuous assignment assigns a value to that signal (write). For example:

```
initial
#5 a = 0; // write operation to signal a
```

```
initial
#5 a = 1; // write operation of signal a
```

In this example, at simulation time 5, different initial blocks assign 0 and 1 to signal a. When simulation time 5 is over you do not know if signal a's value is 0 or 1.

Finding these race conditions is important because in Verilog simulation you cannot control the order of execution of statements in different always or initial blocks or continuous assignments that execute at the same simulation time. This means that a race condition can produce different simulation results when you simulate a design with different, but both properly functioning, Verilog simulators.

Even worse, a race condition can result in different simulation results with different versions of a particular simulator, or with different optimizations or performance features of the same version of a simulator.

Also sometimes modifications in one part of a design can cause hidden race conditions to surface even in unmodified parts of a design, thereby causing different simulation results from the unmodified part of the design.

The indications of a race condition are the following:

- When simulation results do not match when comparing simulators
- When design modifications cause inexplicable results
- When simulation results do not match between different simulation runs of the same simulator, when different versions or different optimization features of that simulator are used

Therefore even when a Verilog design appears to be simulating correctly and you see the results you want, you should look for race conditions and remove them so that you will continue to see the same simulation results from an unrevised design well into the future. Also you should look for race conditions while a design is in development.

VCS can help you find these race conditions by writing report files about the race conditions in your design.

VCS writes the reports at runtime but you enable race detection at compile-time with a compile-time option.

The reports can be lengthy for large designs. You can post-process the report to generate another shorter report that is limited, for example, to only part of the design or to only between certain simulation times.

Enabling Race Detection

When you compile your design you can enable race detection during simulation for your entire design or part of your design.

The `+race` compile-time option enables race detection for your entire design.

The `+racecd` compile-time option enables race detection for the part of your design that is enclosed between the ``race` and ``endrace` compiler directives.

Specifying the Maximum Size of Signals in Race Conditions

You use the `+race_maxvecsize` compile-time option to specify the largest vector signal for which the dynamic race detection tool looks for race conditions. The syntax is as follows:

```
+race_maxvecsize=size
```

For example, if you enter the following `vcs` command line:

```
vcs source.v +race +race_maxvecsize=32
```

This command line specifies running the dynamic race detection tool during simulation and looking for race conditions, of both the read-write and write-write types, for signals with 32 bits or fewer. Notice that the command line still requires the `+race` compile-time option to enable the dynamic race detection tool to start at runtime.

The Race Detection Report

While VCS simulates your design it writes race detection reports to the files `race.out` and `race.unique.out`.

The `race.out` file contains a line for all race conditions it finds at all times throughout the simulation. If VCS executes two different statements in the same time step several times, the `race.out` file contains a line for each of those several times.

The `race.unique.out` file contains lines only for race conditions that are unique, and which have not been reported in a previous line.

Note:

The `race.unique.out` is automatically created by the `PostRace.pl` Perl script after simulation. This script needs a perl5 interpreter. The first line of the script points to perl at a specific location, see ["Modifying the PostRace.pl Script" on page 11-10](#). If that location at your site is not a perl5 interpreter, the script fails with syntax errors.

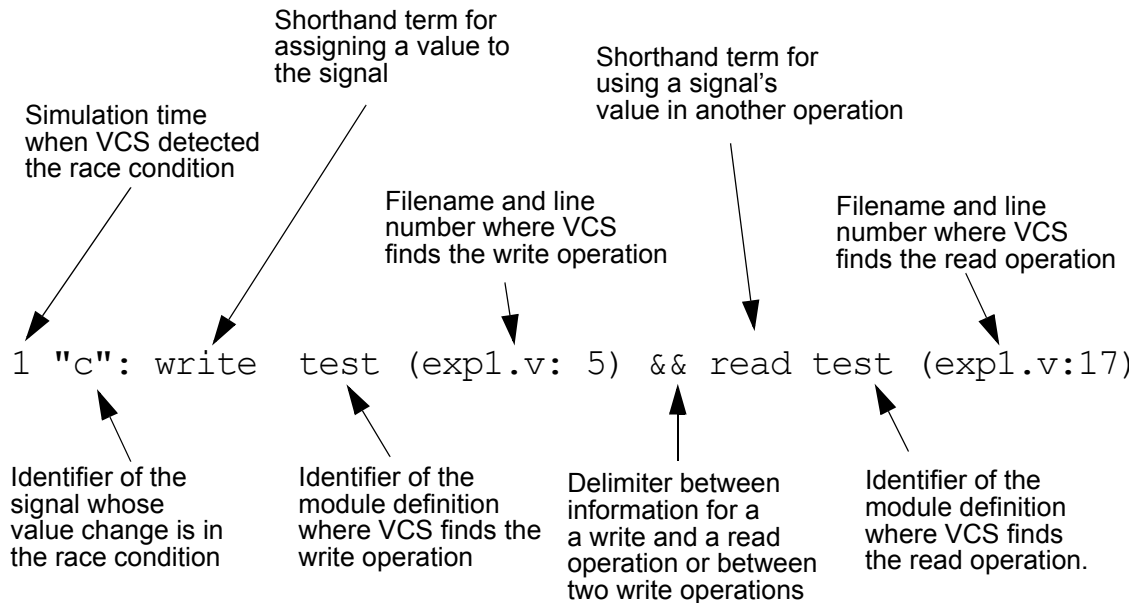
The report describes read-write and write-write race conditions. The following is an example of the contents of a small `race.out` file:

```
Synopsys Simulation VCS RACE REPORT

0 "c": write test (exp1.v: 5) && read test (exp1.v:23)
1 "a": write test (exp1.v: 16) && write test (exp1.v:10)
1 "c": write test (exp1.v: 5) && read test (exp1.v:17)

END RACE REPORT
```

The following explains a line in the `race.out` file:



The following is the source file, with line numbers added, for this race condition report:

```
1. module test;
2. reg a,b,c,d;
3.
4. always @(a or b)
5. c = a & b;
6.
7. always
8. begin
9. a = 1;
10. #1 a = 0;
11. #2;
12. end
13.
14. always
15. begin
16. #1 a = 1;
17. d = b | c;
18. #2;
19. end
20.
21. initial
22. begin
23. $display("%m c = %b",c);
24. #2 $finish;
25. end
26. endmodule
```

As stipulated in race.out:

- At simulation time 0 there is a procedural assignment to reg `c` on line 5 and also `$display` system task displays the value of reg `c` on line 23.
- At simulation time 1 there is a procedural assignment to reg `a` on line 10 and another procedural assignment to reg `a` on line 16.

- Also at simulation time 1 there is a procedural assignment to register `c` on line 5 and the value of register `c` is in an expression that is evaluated in a procedural assignment to another register on line 17.

Races of No Consequence

Sometimes race conditions exist, such a write-write race to a signal at the same simulation time, but the two statements that are assigning to the signal are assigning the same value. This is a race of no consequence and the race tool indicates this with `**NC` at the end of the line for the race in the `race.out` file.

```
0 "r4": write test (nc1.v: 40) && write test
(nc1.v:44)**NC
20 "r4": write test (nc1.v: 40) && write test
(nc1.v:44)**NC
40 "r4": write test (nc1.v: 40) && write test
(nc1.v:44)**NC
60 "r4": write test (nc1.v: 40) && write test (nc1.v:44)
80 "r4": write test (nc1.v: 40) && write test
(nc1.v:44)**NC
```

Post Processing the Report

VCS comes with the `PostRace.pl` Perl script that you can use to post-process the `race.out` report to generate another report that contains a subset of the race conditions in the `race.out` file. You include options on the command line for the `PostRace.pl` script to specify this subset. These options are as follows:

`-hier module_instance`

Specifies the hierarchical name of a module instance. The new report lists only the race conditions found in this instance and all module instances hierarchically under this instance.

`-sig signal`

Specifies the signal that you want to examine for race conditions. You can only specify one signal and must not include a hierarchical name for the signal. If two signals in different module instances have the same identifier, the report lists race conditions for both signals.

`-minmax min max`

Specifies the minimum, or earliest, simulation time and the maximum, or latest, simulation time in the report

`-nozero`

Omits race conditions that occur at simulation time 0.

`-uniq`

Omits race conditions that also occurred earlier in the simulation. The output is the same as the contents of the race.unique.out file.

`-f filename`

Specifies the name of the input file. Use this option if you changed the name of the race.out file

`-o filename`

The default name of the output file is race.out.post. If you want a different name, specify it with this option.

You can enter more than one of these options on the PostRace.pl command line.

If you enter an option more than once, the script uses the last of these multiple entries.

The report generated by the PostRace.pl script is in the race.out.post file unless you specify a different name with the `-o` option.

The following is an example of the command line:

```
PostRace.pl -minmax 80 250 -f mydesign.race.out -o  
mydesign.race.out.post
```

In this example the output file is named `mydesign.race.out.post` and reports on the race conditions between 80 and 250 time units. The post-process file is named `mydesign.race.out`.

Modifying the PostRace.pl Script

The first line of the `PostRace.pl` Perl script is as follows:

```
#!/usr/local/bin/perl
```

If Perl is installed at a different location at your site you need to modify the first line of this script. This script needs a perl5 interpreter. You will find this script at: `vcs_install_dir/bin/PostRace.pl`

Debugging Simulation Mismatches

A design can contain several race conditions where many of them behave the same in different simulations so they are not the cause of a simulation mismatch. For a simulation mismatch you want to find the “critical races,” the race conditions that cause the simulation mismatch. This section describes how to do this.

Add system tasks to generate VCD files to the source code of the simulations that mismatch. Recompile them with the `+race` or `+racecd` options and run the simulations again.

When you have two VCD files, find their differences with the `vcdiff` utility. This utility is located in the `vcs_install_dir/bin` directory. The command line for `vcdiff` is as follows:

```
vcdiff vcdfile1.dmp vcdfile2.dmp -options > output_filename
```

If you enter the `vcdiff` command without arguments, you see usage information including the options.

Method 1: If the Number of Unique Race Conditions is Small

A unique race condition is a race condition that can occur several times during simulation but only the first occurrence is reported in the `race.unique.out` file. If there aren't many lines in the `race.unique.out` file than the number of unique race conditions is small. If so, for each signal in the `race.unique.out` file:

1. Look in the output file from the `vcdiff` utility. If the signal values are different, you have found a critical write-write race condition.
2. If the signal values are not different, look for the signals that are assigned the value of this signal or assigned expressions that include this signal (read operations).
3. If the values of these other signals are different at any point in the two simulations, note the simulation times of these differences on the other signals, and post process the `race.out` file looking for race conditions in the first signal at around the simulation times of the value differences on the other signals. Specify simulation times just before and just after the time of these differences with the `-minmax` option. Enter:

```
PostRace.pl -sig first_signal -minmax time time2
```

If the `race.out.post` file contains the first signal, then it is a critical race condition and must be corrected.

Method 2: If the Number of Unique Races is Large

If there are many lines in the race.unique.out file then there are a large number of unique race conditions. If so, one method of finding the critical race conditions is to do the following:

1. Look in the output file from the vcdiff utility for the simulation time of the first difference in simulation values.
2. Post process the race.out file looking for races at the time of the first simulation value difference. Specify simulation times just before and just after the time of these differences with the `-minmax` option. Enter:

```
PostRace.pl -minmax time time2
```

3. For each signal in the resulting race.out.post file:
 - a. If the simulation values differ in the two simulations, then the race condition in the race.out.post file is a critical race condition.
 - b. If the simulation values are not different, check the signals that are assigned the value of this signal or assigned expressions that include this signal. If the values of these other signals are different then the race condition in the race.out.post file is a critical race condition.

Method 3: An Alternative When the Number of Unique Race Conditions is Large

1. Look in the output file from the vcdiff utility for the simulation time of the first difference in simulation values.
2. For each signal that has a difference at this simulation time:
 - a. Traverse the signal dependency backwards in the design until you find a signal whose values are the same in both simulations.

b. Look for a race condition on that signal at that time. Enter:

```
PostRace.pl -sig signal -minmax time time2
```

If there is a race condition at that time on that signal, it is a critical race condition.

The Static Race Detection Tool

It is possible for a group of statements to combine to form a loop such that the loop will be executed more than once by other Verilog simulators but only once by VCS. This is a race condition.

These situations come about when level sensitive “sensitivity lists” (event controls that immediately following the `always` keyword in an `always` block and which also do not contain the `posedge` or `negedge` keywords) and procedural assignment statements in these `always` blocks combine with other statements, such as continuous assignment statements or module instantiation statements, to form a potential loop. We have found that these situations do not occur if these `always` blocks contain delays or other timing information, non-blocking assignment statements, or PLI calls through user-defined system tasks.

You start the static race detection tool with the `+race=all` compile-time option (not the `+race` compile-time option).

After compilation the static race detection tool writes the file named `race.out.static` that reports on the race conditions it finds.

The following is a excerpt from a line numbered source code example that shows such an `always` block that combines with other statements to form a loop:

```
35  always @( A or C ) begin
36      D = C;
37      B = A;
38  end
39
40  assign C = B;
```

The race.out.static file from the compilation of this source code follows:

```
Race-[CLF] Combinational loop found
      "source.v", 35: The trigger 'C' of the always block
can cause
      the following sequence of event(s) which can again
trigger
      the always block.
      "source.v", 37: B = A;
      which triggers 'B'.
      "source.v", 40: assign C = B;
      which triggers 'C'.
```

12

Delays and Timing

This chapter covers the following topics:

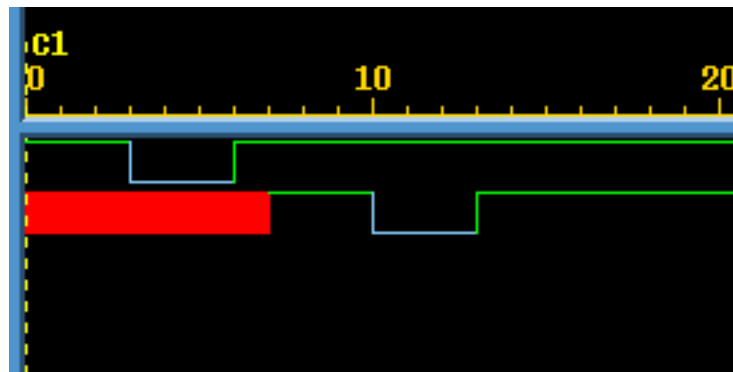
- [Transport and Inertial Delays](#)
- [Pulse Control](#)
- [Specifying the Delay Mode](#)

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 12-1 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 (how this is done is explained in ["Enabling Transport Delays" on page 12-7](#) and ["Pulse Control" on page 12-7](#)).

Figure 12-1 Transport Delay Waveforms

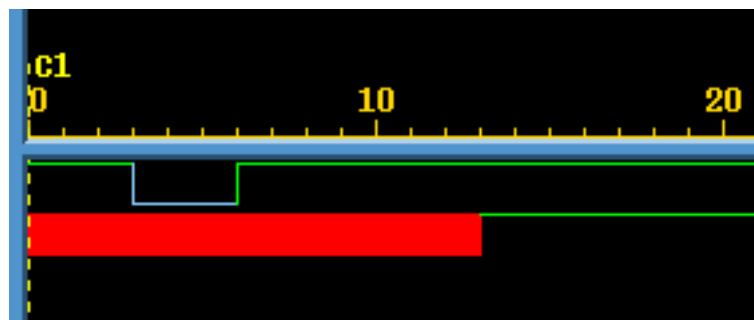


At time 0 a pulse three time units wide begins on the input port. This pulse is narrower than the module path delay of seven time units, but this pulse propagates through the module and appears on the output port after seven time units. Similarly another narrow pulse begins on the input port at time 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 backannotated to a net from an SDF file. For more details on SDF backannotation, see Chapter 15.

Inertial delays, in contrast, filter out all pulses that are narrower than the delay. Figure 12-2 shows the waveforms for the same input and output ports when you have not enabled transport delays for module path delays.

Figure 12-2 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 and inertial delays are the default type of delay for module path delays and INTERCONNECT delays backannotated from an SDF file to a net.

Different Inertial Delay Implementations

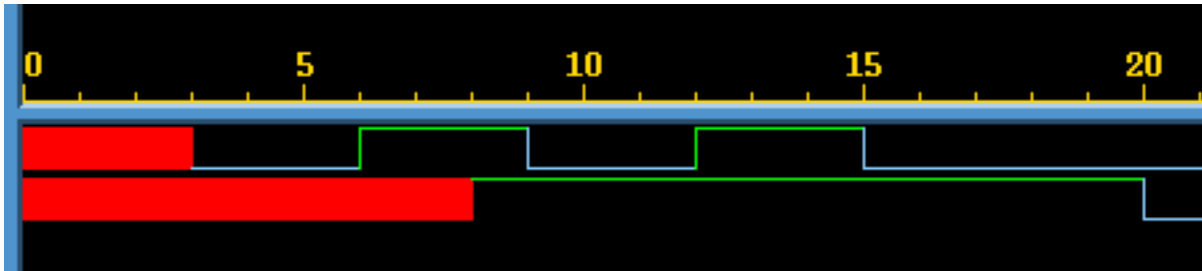
For compatibility with the earlier generation of Verilog simulators, inertial delays have two different implementations, one for primitives (gates, switches and UDPs), continuous assignments, and MIPDs (Module Input Port Delays) and the other for module path delays and INTERCONNECT delays backannotated from an SDF file to a net. For more details on SDF backannotation, see Chapter 15. There is also a third implementation that is for module path and INTERCONNECT delays and pulse control, see ["Pulse Control" on page 12-7](#).

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 12-3 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 12-3 Gate Terminal Waveforms



In the example illustrated in Figure 12-3, 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 that 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, 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. See ["INTERCONNECT Delays" on page 13-32](#).
- 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 12-7](#).

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 12-7](#).

You can specify transport delays on a net to which you backannotate 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 12-7](#).

The `+pulse_e/number`, `+pulse_r/number`, `+pulse_int_e/number`, and `+pulse_int_r/number` options define specific thresholds for pulse width, which let you tell VCS to filter out only some of the pulses and let the other pulses through. See ["Pulse Control" on page 12-7](#).

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 backannotated 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 12-9](#)) or inertial delays (see ["Pulse Control with Inertial Delays" on page 12-12](#)).

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 Pulse on Detect Behavior" on page 12-16](#).

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.

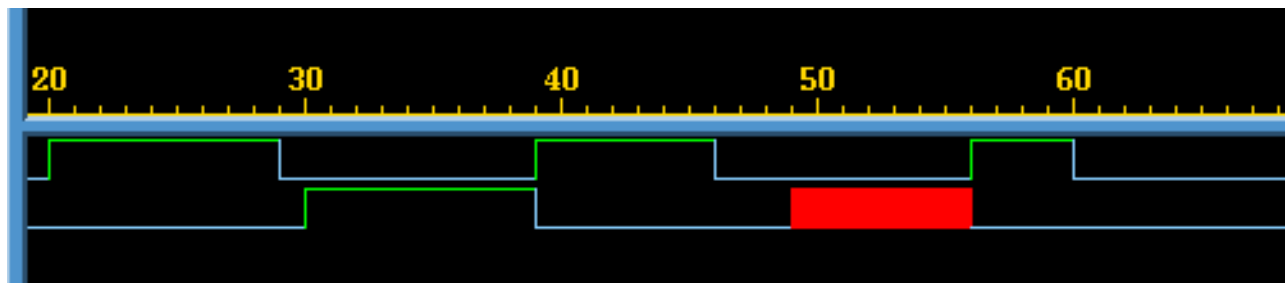
You specify transport delays for INTERCONNECT delays on nets with the `+transport_int_delays`, `+pulse_int_e/number`, and `+pulse_int_r/number` options. You must include all three of these options.

If you want VCS to propagate all pulses, no matter how narrow, specify a 0 percentage. If you want VCS to, for example, replace pulses that are narrower than 80% of the delay with a 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 12-4 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 12-4 Pulse Control with Transport Delays



In the example illustrated in Figure 12-4 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 unit is less than 80% of the module path delay. You also see at this time the following warning message:

```
Warning : Time = 49; Pulse flagged as an error in  
module_instance_name, value = StE.  
Path: input_port --->output_port = 10;
```

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 so 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. When you do 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` number option, the value change on the input propagates to the output.
- If the pulse is not wider than percentage specified by the `+pulse_e/number` 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 percentage specified by the `+pulse_r/number` option, the pulse is filtered out.

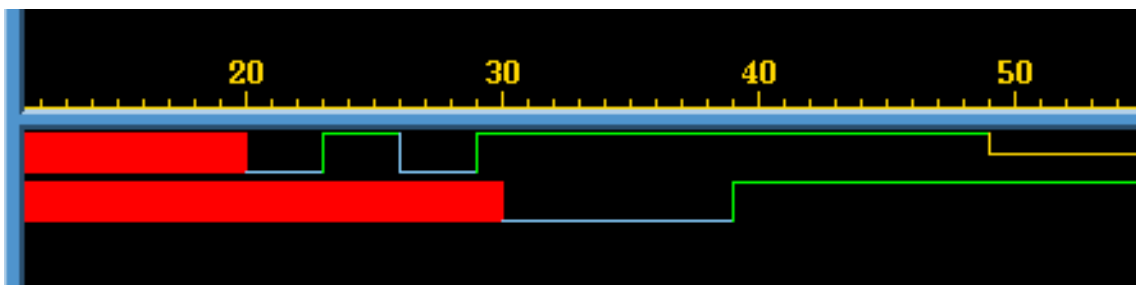
The second scheduled event is always tentative. If another event occurs on the input before the first event occurs on the output, that additional event on the input cancels the second scheduled event and schedules a new second event.

Figure 12-5 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
```

Specifying 0 percentages here 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 12-5 Pulse Control with Inertial Delays



In the example illustrated in Figure 12-5 the following occurs:

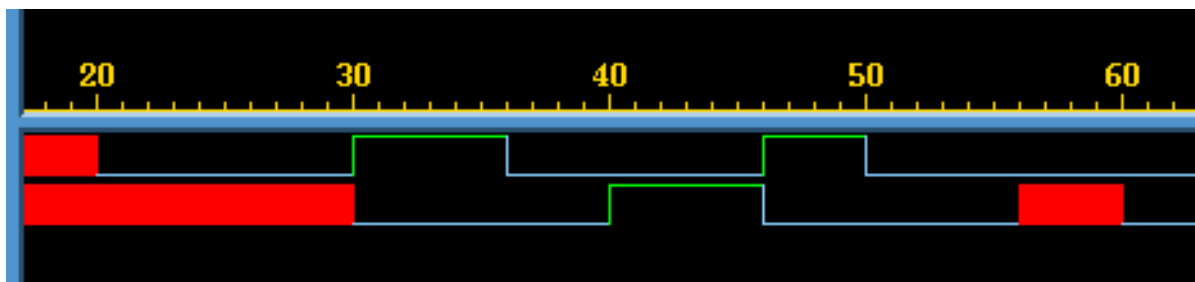
1. At time 20 the input port transitions to 0. This schedules a transition to 0 on the output port at time 30, ten time units later as specified by the module path delay. This is the first scheduled event on the output port. This event is not tentative, it will occur.
2. At time 23 the input port toggles 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 12-6 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 12-6 Pulse Control with Inertial Delays and a Narrow Pulses



In the example illustrated in Figure 12-6 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 so 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 displays the error message:

```
Warning: time = 56; Pulse Flagged as error in
```

```
module_instance_name, Value = StE
port_name ---> port_name = 10;
```

8. At time 60 the output transitions to 0.

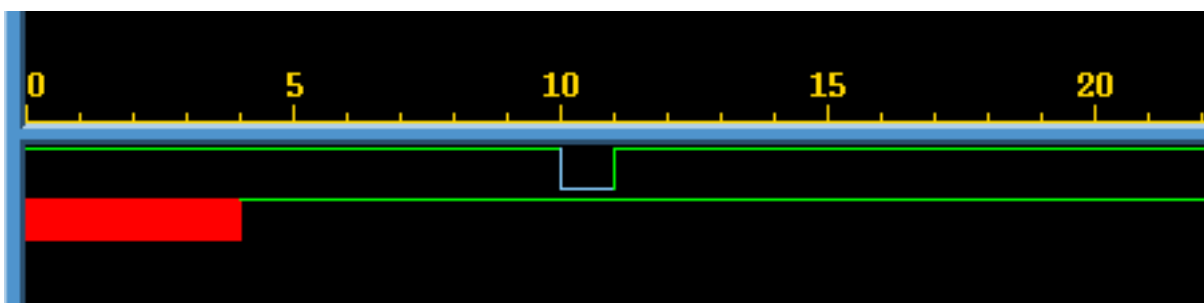
Pulse control sometimes blurs the distinction between inertial and transport delays. In this example the results would have been the same if you also included the `+transport_path_delays` option.

Specifying Pulse on Event or Pulse on 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 12-7.

Figure 12-7 Asymmetric Delays and Scheduling Problems



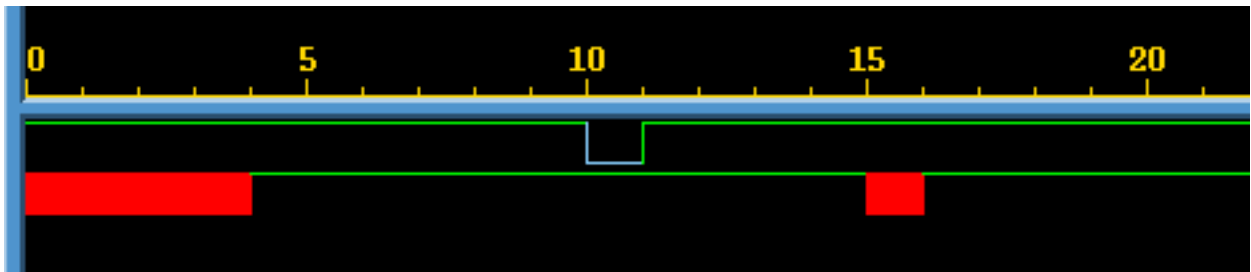
Here you include the `+pulse_e/100` and `+pulse_r/0` options. The scheduling problem is that the leading edge of the pulse on the input, at time 10, schedules a transition to 0 on the output at time 16; but the trailing edge, at time 11, schedules a transition to 1 on the output at time 15.

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 12-8. 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.

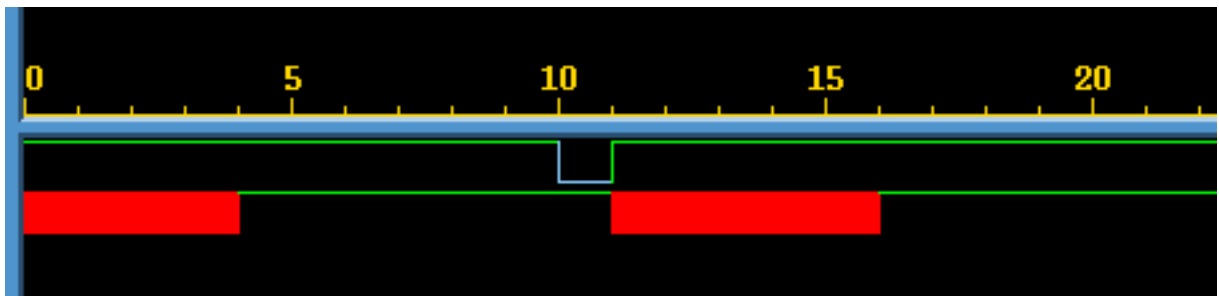
Figure 12-8 Using `+pulse_on_event`



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 12-9 illustrates pulse on detect behavior.

Figure 12-9 Using `+pulse_on_detect`

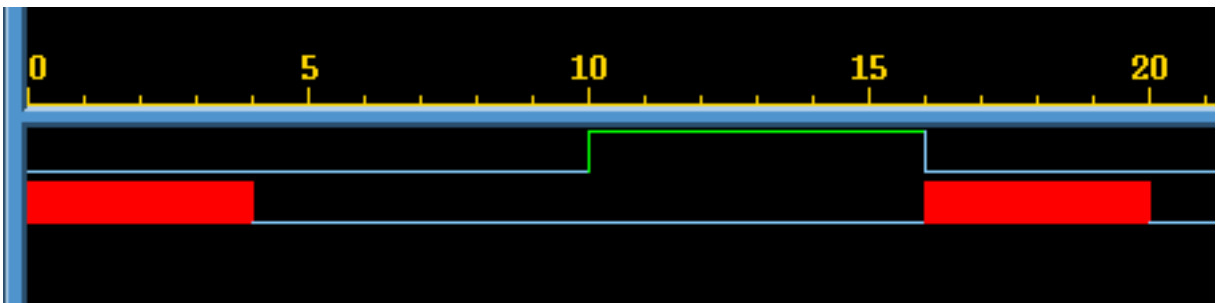


In this example, by including the `+pulse_on_detect` option, VCS causes the leading edge of the X value pulse on the output to begin at time 11 because of an unusual event that occurred on the output between times 15 and 16 because of the rise at simulation time 11.

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 12-10 shows the waveforms for the input and output port when you include the `+pulse_on_detect` option.

Figure 12-10 Pulse on Detect Behavior Showing Multiple Transitions



In the example illustrated in Figure 12-10 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 12-11 shows such a conflict.

Example 12-11 *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 12-11, 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 12-11, 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 12-11, 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 12-11 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 12-11, the module path delay does not equal the distributed delays along the path, VCS uses the longer of the two.

13

SDF Backannotation

This chapter covers the following topics:

- [Using SDF Files](#)
- [Compiling the ASCII SDF File at Compile-Time](#)
- [Reading the ASCII SDF File During Runtime](#)
- [INTERCONNECT Delays](#)
- [Min:Typ:Max Delays](#)
- [Using the Configuration File to Disable Timing](#)
- [Using the timopt Timing Optimizer](#)
- [Editing the timopt.cfg File](#)

Using SDF Files

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 “backannotates” delay values to the design, that is, it adds delay values or changes the delay values specified in the source files. You tell VCS to backannotate delay values with the `$sdf_annotate` system task.

There are two methods that you can use to backannotate delay values from an SDF file:

- Compiling the SDF file at compile time.
This method is preferable in almost all cases and VCS does this by default.
- Reading the ASCII SDF file at run time
This method remains chiefly for compatibility purposes.

Compiling the SDF file, when you compile your Verilog source files, creates binary data files that VCS reads when it executes a `$sdf_annotate` system task at runtime. VCS reads binary data files much faster than ASCII SDF files. The additional compile time will always be less than the time saved at run time. There are, however, limitations on your design when you compile an SDF file. If you cannot circumvent these limitations you can use the method of telling VCS to read the ASCII SDF file when it executes a `$sdf_annotate` system task.

When you use an SDF file to backannotate delay values you can also use an SDF configuration file. In this file you can specify, among other things, the selection of minimal, typical, or maximal delay values in min:typ:max delay value triplets, and the scaling of these delay values. You can then specify these delay value operations for your entire design and on a module by module basis.

Compiling the ASCII SDF File at Compile-Time

At compile time, VCS automatically compiles the SDF file you specify as the first argument to the `$sdf_annotate` system task in your design.

This method saves you simulation time. However, in some cases you may need to disable the automatic compilation of SDF files with the `+oldsdf` compile-time option.

The `$sdf_annotate` System Task

You use the `$sdf_annotate` system task to tell VCS to backannotate delay values from an SDF file to your Verilog design.

The syntax for the `$sdf_annotate` system task is as follows:

```
$sdf_annotate ("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 backannotation starts. The default is the scope of the module instance that calls `$sdf_annotate`.

`"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 backannotation 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 backannotation. You can also use the `+sdfverbose` runtime option to enable the display of all backannotation messages

`"mtm_spec"`

Specifies which delay values of min:typ:max triplets VCS backannotates. 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).

Limitations on Compiling the SDF File

VCS cannot compile your SDF file in the following situations:

- When you do not use a string literal to specify the SDF file in the `$sdf_annotate` system task, for example, when you assign the SDF file name to a register and enter the register as the first argument to the `$sdf_annotate` system task.
- When you include the `scale_type`, or `scale_factor` arguments in the `$sdf_annotate` system task.

If your design contains either of these situations, you must use the method of reading the ASCII SDF file during runtime and include the `+oldsdf` compile-time option.

Example 13-1 Compiling the SDF File Example

The following Verilog model, in source file `ex.v`, does not contain either of these situations:

```
`timescale 1ns / 1ns

module test();
wire in, out, clk, out1;
    initial $sdf_annotate( "./ex.sdf");
    leafA leaf1(out, in, clk);
    leafB leaf2(out1, out, clk);
endmodule

module leafA(out,D,CK);
output out;
```

```

input CK, D;

specify
  (D *> out) = (1,2);
  (CK *> out) = (3);
endspecify
endmodule

module leafB(out,D,CK);
output out;
input D;
input CK;

buf(out,D);

endmodule

```

The following is the SDF file, `ex.sdf`, for the Verilog model.

```

(DELAYFILE
  (DESIGN          "test")
  (VENDOR          "")
  (DIVIDER         ".")
  (VOLTAGE         ":1:")
  (PROCESS         "typical")
  (TEMPERATURE:1:)
  (TIMESCALE1ns)
  (CELL
    (CELLTYPE      "leafB")
    (INSTANCE      leaf2)
    (DELAY
      (ABSOLUTE
        (PORT D (1:2:3)))
      )
    )
  )
  (CELL
    (CELLTYPE      "leafA")
    (INSTANCE      leaf1)
    (DELAY
      (ABSOLUTE
        (IOPATH  D out (7)))
      )
    )
  )
)

```

```
)  
)  
)
```

The following is the vcs command line that compiles both the Verilog source file and the SDF file:

```
vcs +compsdf ex.v
```

You do not have to specify the `ex.sdf` file, or any SDF table file, on the vcs command line. When VCS compiles the SDF file it creates binary data files in the `simv.daidir` directory. VCS reads these binary files when it executes the `$sdf_annotate` system task.

Precompiling an SDF File

Whenever you compile your design, if your design backannotates SDF data, VCS parses either the ASCII text SDF file or the precompiled version of the ASCII text SDF file that VCS can make from the original ASCII text SDF file. VCS does this even if the SDF file is unchanged and already compiled into a binary version by a previous compilation, and even when you are using incremental compilation and the parts of the design backannotated by the SDF file are unchanged.

VCS can parse the precompiled SDF file much faster than it can parse the ASCII text SDF file, so for large SDF files it's a good idea to have VCS create a precompiled version of the SDF file.

Creating the Precompiled Version of the SDF file

To create the precompiled version of the SDF file, include the `+csdf+precompile` option on the vcs command line.

By default the `+csdf+precompile` option creates the precompiled SDF file in the same directory as the ASCII text SDF file and differentiates the precompiled version by appending "_c" to its extension. For example, if the `/u/design/sdf` directory contains a `design1.sdf` file, using the `+csdf+precompile` option creates the precompiled version of the file named `design1.sdf_c` in the `/u/design/sdf` directory.

After you have created the precompiled version of the SDF file you no longer need to include the `+csdf+precompile` option on the `vcs` command line unless there is a change in the SDF file. Continuing to include it, however, such as in a script that you run every time you compile your design, would have no effect when the precompiled version is newer than the ASCII text SDF file, but would create a new precompiled version of the SDF file whenever the ASCII text SDF file changes. Therefore this option is intended to be used in scripts for compiling your design.

When you recompile your design, VCS finds the precompiled SDF file in the same directory as the SDF file specified in the `$sdf_annotate` system task. You can also specify the precompiled SDF file in the `$sdf_annotate` system task.

Specifying an Alternative Name and Location

You can use the `+csdf+precomp+dir+directory` option to specify the directory path where you want VCS to write the precompiled SDF file. When you do, make sure that you have already created all directories in the path because VCS does not create directories that are specified with this option.

You can use the `+csdf+precomp+ext+ext` option to specify an alternative to the "_c" character string addition to the filename extension of the precompiled SDF file.

For example, in the /u/designs/mydesign directory are the design.v and design.sdf files and the sdfhome directory. If you enter the following command line:

```
vcs design.v +csdf+precompile +csdf+precomp+dir+sdfhome
+csdf+precomp+ext+_precompiled
```

VCS creates the design.sdf_precompiled file in the sdfhome directory.

Now that the precompiled file is not in the default location and does not have the default filename extension, in subsequent compilations you must tell VCS its new location and name. There are two ways to do this:

1. Continue to include the location and name options on the vcs command line in subsequent compilations. In this example you would always include +csdf+precomp+dir+sdfhome and +csdf+precomp+ext+_precompiled.

This method does not require you to make a change in the source code. You can just add these options to a script you run whenever you compile your design.

2. Change the filename argument in the \$sdf_annotate system task to the precompiled file. In this example you would change:

```
$sdf_annotate("design.sdf");
```

to:

```
$sdf_annotate("sdfhome/design.sdf_precompiled");
```

Reading the ASCII SDF File During Runtime

You can use the ACC capability support that is part of the PLI interface to tell VCS to read the ASCII SDF file when it executes the `$sdf_annotate` system task.

To do this, include the `+oldsdf` compile-time option and create a PLI table file that maps the `$sdf_annotate` system task to the C function `sdf_annotate_call` (automatically provided with VCS) and indicates which modules will be annotated and what types of constructs will be annotated.

For faster simulation enable, in the PLI table, only the ACC capabilities you need. These capabilities are as follows:

`tchk:`

Annotate to timing checks (sdf SETUP, HOLD, etc.)

`gate:`

Annotate to gate primitives (sdf DEVICE)

`mp:`

Annotate propagation delays to module paths (sdf IOPATH)

`mip:`

Annotate propagation delays to module input port delays (sdf PORT and INTERCONNECT)

`mipb:`

Annotate to module input port bit delays (sdf PORT and INTERCONNECT)

`prx:`

Annotate pulse rejection and error delays to module paths (sdf PATHPULSE)

For example, the SDF annotation of module paths contained within the hierarchy of a module named `myasic` requires a PLI table such as this:

```
$sdf_annotate call=sdf_annotate_call acc+=mp,prx:myasic+
```

If possible, take advantage of the `%CELL` wildcard scope to add the needed ACC capabilities to library cells only, which are often the only cells requiring SDF annotation:

```
$sdf_annotate call=sdf_annotate_call acc+=mp, prx:%CELL
```

For methodologies requiring delay annotation in the sub-hierarchy of cells, use:

```
$sdf_annotate call=sdf_annotate_call acc+=mp, prx:%CELL+
```

When running `vcs` use the `-P` compile-time option to specify this PLI table, as you would to specify any user task or function implemented in a custom PLI application. You do not need to provide the function `sdf_annotate_call` as it is part of the VCS product by default.

```
% vcs -P sdf.myasic.tab myasic.v +oldsdf
```

Example 13-2 Reading the ASCII SDF File Example

The following Verilog model, in source file `ex2.v`, contains a `specparam` in its `specify` block:

```
`timescale 1 ns / 1 ns
module top;
reg in;
leaf leaf1(in, out);
initial begin
    $sdf_annotate("ex2.sdf", top);
    $monitor($time, , in, , out);
    in = 0;
    #100 $finish;
end
```

```

end
endmodule
module leaf(in,out);
input in;
output out;
buf(out,in);
specify
    specparam mpath_d=1.0;
    (in => out) = (mpath_d);
endspecify
endmodule

```

The following is the SDF file, `ex2.sdf`, for the Verilog model:

```

(DELAYFILE
  (TIMESCALE 1 ns)
    (CELL
      (CELLTYPE "leaf")
      (INSTANCE leaf1)
      (DELAY
        (ABSOLUTE
          (IOPATH in out (5))))))

```

In this file the SDF construct `IOPATH` corresponds to a Verilog module path delay. The delay value specified is 5. The time unit, specified by the `TIMESCALE` construct, makes the annotated delay value to the module path delay 5 ns.

The PLI table file, `ex2.tab` contains the following:

```

$sdf_annotate call=sdf_annotate_call acc=mp:top+

```

We specify the PLI table file on the `vcs` command line:

```

vcs -P ex2.tab ex2.v

```

We see the successful backannotation of the delay value when we execute the `simv` executable and see transition times and values from the `$monitor` system task:


```
0 0 x
5 0 0
$finish at simulation time 100
V C S           S i m u l a t i o n           R e p o r t
Time: 100 ns
CPU Time: 0.100 seconds; Data structure size: 0.0Mb
```

Performance Considerations

Because the compiler must make large quantities of information about the structure of the design available at run time in order to allow annotation, you must consider simulation efficiency when using SDF annotation. Keep in mind the following:

- For annotation capabilities `gate`, `mp`, and `tchk`, there is overhead only if the modules actually contain the related constructs.
- For module input port delays, significant compile and runtime overhead can be incurred if annotation is enabled on ports that will not be annotated.
- Enabling port bit annotation increases the overhead.

Use the `%CELL` wildcard scope as shown in the previous section (or `%CELL+` if all the annotated cells are below the `$sdf_annotate` call) to annotate only those modules that require SDF annotation ACC capabilities.

Replacing Negative Module Path Delays in SDF Files

VCS does not backannotate negative module path delays from IOPATH entries in SDF files. By default VCS substitutes a 0 delay for these negative delays. You can tell VCS to instead use the module path delay specified in a module's `specify` block by including the `+old_iopath` compile-time option.

Using the Shorter Delay in IOPATH Entries

It is valid syntax in an SDF file to have two IOPATH entries for the same pair of ports but one entry for a rising edge on the input or inout port and another entry for a falling edge on the input or inout port. For example:

```
(IOPATH (posedge inport) output (3) (6))  
(IOPATH (negedge inport) output (5) (4))
```

These entries specify backannotating the following delay values to the module path delay between the port named inport and the port named output:

- If a rising edge on inport results in a rising edge on output, then the module path delay is three.
- If a rising edge on inport results in a falling edge on output, then the module path delay is six.
- If a falling edge on inport results in a rising edge on output, then the module path delay is five.
- If a falling edge on inport results in a falling edge on output, then the module path delay is four.

VCS does not however backannotate the module path delay values as specified. Instead VCS backannotates the shorter of the corresponding delays in the two IOPATH entries. Therefore the following behavior occurs during simulation when a value propagates from inport to output:

- When the value propagation results in a rising edge on output, the module path delay from inport to output is three, no matter whether the value propagation began with a rising or falling edge on inport, because three is shorter than five.
- When the value propagation results in a falling edge on output, the module path delay from inport to output is four, no matter whether the value propagation began with a rising or falling edge on inport, because four is shorter than six.

In this example there are two delay values in the two IOPATH entries for the same pair of ports, for a rising and falling edge on the output or inport, but VCS would also backannotate the shorter of the corresponding delay values if the IOPATH entries each had only one delay value or each had three or six delay values (delay value lists with 12 values are not implemented).

VCS does this, backannotates the shorter of the corresponding delay values, to be compatible with the earlier generation of Verilog simulators, Cadence's Verilog-XL.

Disabling CELLTYPE Checking in SDF Files

Sometimes when you merge smaller designs into a larger design you discover that you have more than one module in the larger design with the same identifier or name. When this happens you must resolve the conflict by editing your Verilog source code to rename one of the modules that shares the same identifier.

If an SDF file backannotates delay values to an instance of a module that must be renamed, the CELLTYPE entry in the SDF file for the instance will contain the old identifier of the module, and ordinarily this would cause VCS to display the following error message:

```
SDF Error: Instance instance_name is CELLTYPE of
"new_module_identifier" not "old_module_identifier",
ignoring
```

In this situation, to avoid editing the SDF file, include the `+sdf_nocheck_celltype` compile-time option on the `vcs` command line.

The SDF Configuration File

You can use the configuration file to control the following on a module type basis as well as a global basis:

- min:typ:max selection
- Scaling
- Turnoff delay determination
- MIPD (module-input-delay) approximation policy for cases of 'overlapping' annotations to the same input port.

Additionally, there is a mapping command you can use to redirect the target of `IOPATH` and `TIMINGCHECK` statements from the scope of the `INSTANCE` to a specific `IOPATH` or `TIMINGCHECK` in its subhierarchy for all instances of a specified module type.

Delay Objects and Constructs

The mapping from SDF statements to simulation objects in VCS is fixed, as shown in Table 13-1.

Table 13-1 VCS Simulation Delay Objects/Constructs

	SDF Constructs	VCS Simulation Object
Delays	PATHPULSE	module path pulse delay
	GLOBALPATHPULSE	module path pulse reject/error delay
	IOPATH	module path delay
	PORT	module input port delay
	INTERCONNECT	module input port delay or, intermodule path delay when <code>+multisource_int_delays</code> specified
	NETDELAY DEVICE	module input port delay primitive and module path delay
Timing-checks	SETUP	<code>\$setup</code> timing-check limit
	HOLD	<code>\$hold</code> timing-check limit
	SETUPHOLD	<code>\$setup</code> and <code>\$hold</code> timing-check limit
	RECOVERY	<code>\$recovery</code> timing-check limit
	SKEW	<code>\$skew</code> timing-check limit
	WIDTH	<code>\$width</code> timing-check limit
	PERIOD	<code>\$period</code> timing-check limit
	NOCHANGE	ignored
	PATHCONSTRAINT	ignored
	SUM	ignored
	DIFF	ignored

Table 13-1 VCS Simulation Delay Objects/Constructs

SDF Constructs	VCS Simulation Object
SKEWCONSTRAINT	ignored

SDF Configuration File Commands

This section explains the commands used in SDF configuration files, with syntax and examples.

approx_command:

The `INTERCONNECT_MPID` keyword selects the INTERCONNECT delays in the SDF file that are mapped to MIPDs in VCS. It can specify one of the following to VCS:

MINIMUM

Annotates, to the MIPD for the input or inout port instance, the shortest delay of all the INTERCONNECT delay value entries in the SDF file that specify a connection to the input or inout port.

MAXIMUM

Annotates, to the MIPD for the input or inout port instance, the longest delay of all the INTERCONNECT delay value entries in the SDF file that specify a connection to the input or inout port.

AVERAGE

Annotates, to the MIPD for the input or inout port instance, the average delay of all the INTERCONNECT delay value entries in the SDF file that specify a connection to the input or inout port.

LAST

Annotates, to the MIPD for the input or inout port instance, the delays in the last INTERCONNECT entry in the SDF file that specifies a connection to the input or inout port.

The default approximation is MAXIMUM.

Syntax:

```
INTERCONNECT_MIPD = MINIMUM | MAXIMUM | AVERAGE | LAST;
```

Example:

```
INTERCONNECT_MIPD=LAST;
```

map_command:

This command maps the SDF constructs to VCS Simulation Delay Objects.

Note:

Refer to [Table 13-1: VCS Simulation Delay Objects/Constructs](#).

Syntax:

```
sdf_construct = veritool_map ;  
sdf_construct : IOPATH | PORT | INTERCONNECT | NETDELAY |  
DEVICE | SETUP | HOLD | SETUPHOLD | RECOVERY | SKEW | WIDTH  
| PERIOD | NOCHANGE | PATHPULSE | GLOBALPATHPULSE  
veritool_map : IGNORE | INTERMOD_PATH | MIPD | CELL | USE
```

Example:

```
INTERCONNECT=MIPD;  
PATHPULSE=IGNORE;
```

mtm_command:

Annotates the minimum, typical, or maximum delay value. Specifies one of the following keywords:

MINIMUM

Annotates the minimum delay value

TYPICAL

Annotates the typical delay value

MAXIMUM

Annotates the maximum delay value

TOOL_CONTROL

Delay value is determined by the command line options of the Verilog tool (+mindelays, +typdelays, or +maxdelays)

The default for min_typ_max is TOOL_CONTROL.

Syntax:

```
MTM = MINIMUM | TYPICAL | MAXIMUM | TOOL_CONTROL;
```

Example:

```
MTM=MAXIMUM;
```

scale_command:

- SCALE_FACTORS - Set of three real number multipliers that scale the timing information in the SDF file to the minimum, typical, and maximum timing information that is backannotated to the Verilog tool. The multipliers each represent a positive real number, for example 1.6:1.4:1.2
- SCALE_TYPE - Selects one of the following keywords to scale the timing specification in the SDF file to the minimum, typical, and maximum timing that is backannotated to the Verilog tool:

FROM_MINIMUM

Scales from the minimum timing specification in the SDF file.

FROM_TYPICAL

Scales from the typical timing specification in the SDF file.

FROM_MAXIMUM

Scales from the maximum timing specification in the SDF file.

FROM_MTM

Scales directly from the minimum, typical, and maximum timing specifications in the SDF file.

Syntax:

```
SCALE_FACTORS = number : number : number ;  
SCALE_TYPE = FROM_MINIMUM | FROM_TYPICAL | FROM_MAXIMUM |  
FROM_MTM ;
```

Example:

```
SCALE_FACTORS=100:0:9;  
SCALE_TYPE=FROM_MTM;  
SCALE_FACTORS=1.1:2.1:3.1;  
SCALE_TYPE=FROM_MINIMUM;
```

turnoff_command:

The `turnoff_command` keyword globally specifies which delays the SDF annotator uses. It uses one of the following for a given object:

MINIMUM

Minimum of the rise and fall delay value.

MAXIMUM

Maximum of the rise and fall delay value.

AVERAGE

Average of the rise and fall delay value.

FROM_FILE

The SDF annotator uses the turn-off delays in the SDF file. If you do not specify `FROM_FILE` or if you specify `FROM_FILE`, but the SDF files does not contain the turn-off delay, the turn-off delay is set to min (rise,fall).

Syntax:

```
TURNOFF_DELAY = FROM_FILE | MINIMUM | MAXIMUM | AVERAGE ;
```

Example:

```
TURNOFF_DELAY=FROM_FILE;
```

modulemap_command:

Redirects the delay object of IOPATH or TIMINGCHECK SDF statements for all instances of a specified module type to a different module path or timing check object in the same or lower scope.

Syntax:

```
MODULE verilog_id { list_of_module_mappings } ;
```

Here, *verilog_id* is the name of a specific type of module (not instance name) specified in the corresponding Verilog description.

list_of_module_mappings:

```
mtm_command | scale_command | map_inner_command
```

The mtm and scale commands are as defined earlier. Note that using these commands as arguments for the module_map_command affects only the IOPATH, DEVICE, and TIMINGCHECK information annotated to a specific type of module.

Syntax:

```
MAP_INNER = verilog_id ;  
| systchk = ADD { list_of_systchk }  
| systchk = ADD { list_of_systchk }  
| systchk = OVERRIDE { list_of_systchk }  
| systchk = IGNORE ;  
| path_declaration = ADD { list_of_path_declaration }  
| path_declaration = OVERRIDE { list_of_path_declaration }
```

```
| path_declaration = IGNORE ;
```

The SDF annotator uses `hierarchical_path` as the Verilog hierarchical path name of a submodule within `module_type`. The paths specified in the SDF file are mapped to `module_type`. This path applies to all path delays and timing checks specified for this module in the SDF file including those mapped with `ADD` and `OVERRIDE`.

`ADD`

Adds to the mapping specifications of the SDF file. The original timing specification is mapped to `new_timing`, the Verilog HDL syntax of a path delay or timing check.

`OVERRIDE`

Replaces the mapping specifications of the SDF file. The original timing specification is mapped to `new_timing`, the Verilog HDL syntax of a path delay or timing check.

`IGNORE`

Ignores the mapping specifications in the SDF file. In all cases, the `hierarchical_path` name is applied to any `new_timing` specification before they are annotated to VCS.

```
list_of_systchk : systchk ';' | list_of_systchk systchk ';'
systchk: '$setup' '(' systchk_arg ',' systchk_arg ',' expression opt_notifier ')'
| '$hold' '(' systchk_arg ',' systchk_arg ',' expression
  opt_notifier ')'
| '$setuphold' '(' systchk_arg ',' systchk_arg ',' expression ','
  expression opt_notifier ')'
| '$recovery' '(' systchk_arg ',' systchk_arg ',' expression
  opt_notifier ')'
| '$period' '(' systchk_arg ',' expression opt_notifier ')'
| '$width' '(' systchk_arg ',' expression ',' expression
  opt_notifier ')'
| '$skew' '(' systchk_arg ',' systchk_arg ',' expression
  opt_notifier ')'
| '$nochange' '(' systchk_arg ',' systchk_arg ',' expression ','
  expression opt_notifier ')'
opt_notifier: ',' expression | ',' | ;
systchk_arg: expression
| expression '&&&' timing_check_condition
| timing_check_event_control specify_terminal_descriptor
| timing_check_event_control specify_terminal_descriptor
  '&&&' timing_check_condition
```

```

timing_check_condition: expression
list_of_path_declaration: path_declaration ';'
    | list_of_path_declaration path_declaration ';'
path_declaration: opt_if '(' list_of_path_in_td path_type list_of_path_out_td
    ')'
    | opt_if '(' list_of_path_in_td path_type '('
list_of_path_out_td ')' ')'
opt_if: 'if' '(' expression ')' | ;
opt_edge: timing_check_event_control | ;
timing_check_event_control: 'posedge' | 'negedge' | 'edge'
    '[' edge_descriptor_list ']'
edge_descriptor_list: edge_descriptor
    | edge_descriptor_list ',' edge_descriptor
edge_descriptor : '01' | '10' | '0x' | 'x1' | '1x' | 'x0'
path_type: '=>' | '- '=>' | '+ '=>' | '*>' | '- '*>' | '+ '*>'
list_of_path_out_td: list_of_path_out_td ','
    specify_out_terminal_descriptor | specify_out_terminal_descriptor
specify_out_terminal_descriptor: '(' specify_terminal_descriptor data_op
expression ')'
    | specify_terminal_descriptor
    data_op : ':' | '- ':' | '+ ':'
list_of_path_in_td: list_of_path_in_td ','
    opt_edge_specify_terminal_descriptor
    | opt_edge_specify_terminal_descriptor ;
opt_edge_specify_terminal_descriptor : opt_edge
    specify_terminal_descriptor ;
specify_terminal_descriptor: verilog_id | verilog_id '[' expression ']'
    | verilog_id '[' expression ':' expression ']' ;
expression : primary | unary_op primary
    | expression '+' expression | expression '-' expression
    | expression '*' expression | expression '/' expression
    | expression '%' expression | expression '==' expression
    | expression '!=' expression | expression '===' expression
    | expression '!==' expression | expression '&&' expression
    | expression '||' expression | expression '<' expression
    | expression '<=' expression | expression '>' expression
    | expression '>=' expression | expression '&' expression
    | expression '|' expression | expression '^' expression
    | expression '^~' expression | expression '~^' expression
    | expression '>>' expression | expression '<<' expression
    | expression '?' expression ':' expression
unary_op : '!' | '~' | '+' | '-' | '&' | '~&' | '|' | '~|' | '^' | '~^' | '^~'

primary : number | lident | lident '[' number ']'
    | lident '[' number ':' number ']' | '{' cat_expr_list '}'
    | '{' expression '{' cat_expr_list '}' '}' | '(' expression ')'
cat_expr_list: cat_expr_list ',' expression | expression

lident: identifier

```

SDF Backannotation

```

identifier: verilog_id | identifier_head verilog_id

identifier_head : verilog_id '.' | identifier_head verilog_id '.'

number : "Any sized or unsized literal decimal, octal, binary, hex, or real number"

verilog_id : "Any legal escaped or non-escaped Verilog identifier (excluding
range selection portion in square brackets)."
```

Example:

```

MODULE sub      {
    // scale_commands
    SCALE_TYPE=FROM_MTM;
    SCALE_FACTORS=1:2:3;
    // mtm_commands
    MTM=MINIMUM;
    // map_inner_commands
    MAP_INNER = X;
    (i1 *> o1) = IGNORE;
    (i *> o1) = ADD { (ib *> oa); }
    (i1 *> o1) = ADD { (ia *> oa); }
    (i1 *> o1) = ADD { (ia *> oa); }
    (i1 *> o1) = ADD { (ib *> ob); }
    if (i2==1) (i2 *> o2) = ADD { (ib *> ob); }
}

```

SDF Example with Configuration File

The following example uses the VCS SDF configuration file sdf.cfg:

```

// test.v - test sdf annotation
`timescale 1ns/1ps
module test;
initial begin
    $sdf_annotate("./test.sdf",test, "./sdf.cfg",,,,,);
end
wire  out1,out2;
wire  w1,w2;
reg  in;
reg  ctrl,ctrlw;
```

```

sub  Y (w1,w2,in,in,ctrl,ctrl);
sub  W (out1,out2,w1,w2,ctrlw,ctrlw);
initial begin
    $display("  i c ww oo");
    $display("ttt  n t 12 12");
    $monitor($realtime,,,in,,ctrl,,w1,w2,,out1,out2);
end
initial begin
    ctrl = 0; // enable
    ctrlw = 0;
    in = 1'bx; //stabilize at x;
    #100 in = 1; // x-1
    #100 ctrl = 1; // 1-z
    #100 ctrl = 0; // z-1
    #100 in = 0; // 1-0
    #100 ctrl = 1; // 0-z
    #100 ctrl = 0; // z-0
    #100 in = 1'bx; // 0-x
    #100 ctrl = 1; // x-z
    #100 ctrl = 0; // z-x
    #100 in = 0; // x-0
    #100 in = 1; // 0-1
    #100 in = 1'bx; // 1-x
end
endmodule
`celldefine
module sub(o1,o2,i1,i2,c1,c2);
output o1,o2;
input  i1,i2;
input  c1,c2;
bufif0 Z(o1,i1,c1);
bufif0 (o2,i2,c2);
specify
    (i1,c1 *> o1) = (1,2,3,4,5,6);
    // 01 = 1, 10 = 2, 0z = 3, z1 = 4, 1z = 5, z0 = 6
    if (i2==1'b1) (i2,c2 *> o2) = (7,8,9,10,11,12);
    // 01 = 7, 10 = 8, z1 = 10, 1z = 11, z0 = 12
endspecify
subsub  X ();
endmodule
`endcelldefine
module subsub(oa,ob,ib,ia);

```

```

input ia,ib;output oa,ob;
specify
    (ia *> oa) = 99.99;
    (ib *> ob) = 2.99;
endspecify
endmodule

```

SDF File: test.sdf

```

(DelayFile
(SDFVersion "3.0")
(Design "sdfctest")
(Date "July 14, 1997")
(Vendor "Synopsys")
(Program "manual")
(Version "4.0")
(Divider .)
(Voltage )
(Process "")
(Temperature )
(Timescale 1 ns)
(Cell (CellType "sub")
(Instance *)
(Delay (Absolute
(IOPATH i1 o1
(10:11:12) (13:14:15) (16:17:18) (19:20:21) (22:23:24) (25:26:27)
))
(COND (i2==1) (IOPATH i2 o2
(10:11:12) (13:14:15) (16:17:18) (19:20:21) (22:23:24) (25:26:27)))
))
)
)
)

```

SDF Configuration File: sdf.cfg

```

INTERCONNECT=MIPD;
PATHPULSE=IGNORE;
INTERCONNECT_MIPD=MAXIMUM;
MTM=TOOL_CONTROL;
SCALE_FACTORS=100:0:9;
SCALE_TYPE=FROM_MTM;
TURNOFF_DELAY=FROM_FILE;
MTM = TYPICAL;
SCALE_TYPE=FROM_MINIMUM;
SCALE_FACTORS=1.1:2.1:3.1;

```

```

MODULE sub {
    SCALE_TYPE=FROM_MTM;
    SCALE_FACTORS=1:2:3;
    MTM=MINIMUM;
    MAP_INNER = X;
    (i1 *> o1) = IGNORE;
    (i1 *> o1) = ADD { (ia *> oa); }
    (i1 *> o1) = ADD { (ib *> ob); }
    if (i2==1) (i2 *> o2) = ADD { (ib *> ob); }
}

```

Understanding the DEVICE Construct

The `DEVICE` construct specifies the intrinsic delay of a cell or gate. When VCS reads a `DEVICE` construct, it backannotates the pin-to-pin delay generics throughout the cell. How VCS handles the `DEVICE` construct depends on whether or not the construct includes an output port name.

- If `DEVICE` includes an output port name (for example, (`DEVICE Out1 (1:2:3)`)), VCS assigns the timing value only to those pin-to-pin delay generics in the cell with that output port name as their destination. For example, during backannotation of the device in Figure 13-3, if the `DEVICE` construct is

```
DEVICE Out1 (1:2:3) (1:2:3)
```

VCS assigns the timing value only to the following generics:

```
tpdIn1_Out1_R, tpdIn1_Out1_F, tpdIn2_Out1_R, and
tpdIn2_Out1_F
```

It ignores the generics such as `tpdIn1_Out2_R`, `tpdIn1_Out2_F`, `tpdIn2_Out2_R`, and `tpdIn2_Out2_F`.

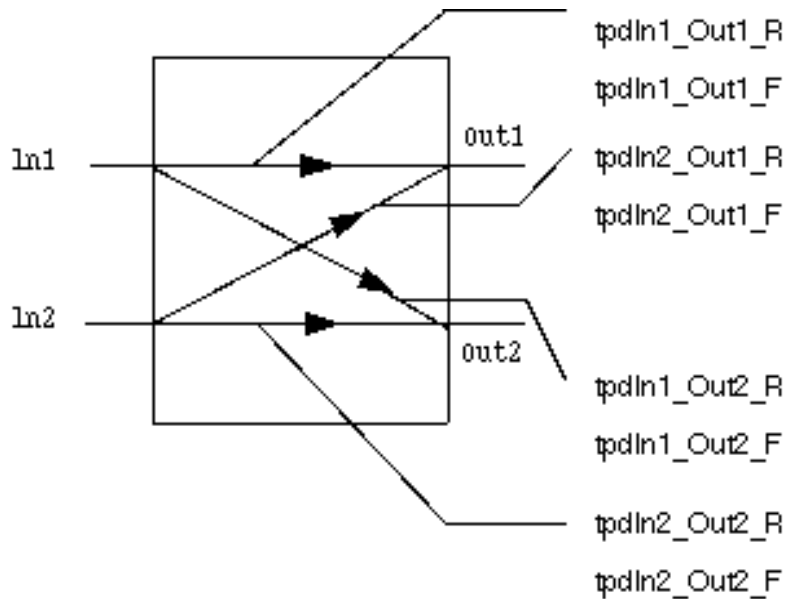
- If `DEVICE` does not include an output port name, VCS assigns the value to all the pin-to-pin delay (`tpd`) generics in the cell. For example, if the `DEVICE` construct is

```
DEVICE (1:2:3)
```

VCS assigns the timing value to all the following generics:

```
tpdIn1_Out1_R, tpdIn2_Out1_R, tpdIn1_Out2_R,  
tpdIn2_Out2_R, tpdIn1_Out1_F, tpdIn2_Out1_F,  
tpdIn1_Out2_F, and tpdIn2_Out2_F.
```

Figure 13-3 Cell with Four Possible pin-to-pin Timing Arcs



Handling Backannotation to I/O Ports

SDF Reader might incorrectly handle the `DEVICE` construct for an output I/O port if the `DEVICE` construct does not include an output port name. Consider Figure 13-4; Port B is an I/O port of the instance U1 of type `IO_CELL`. If the `DEVICE` construct is

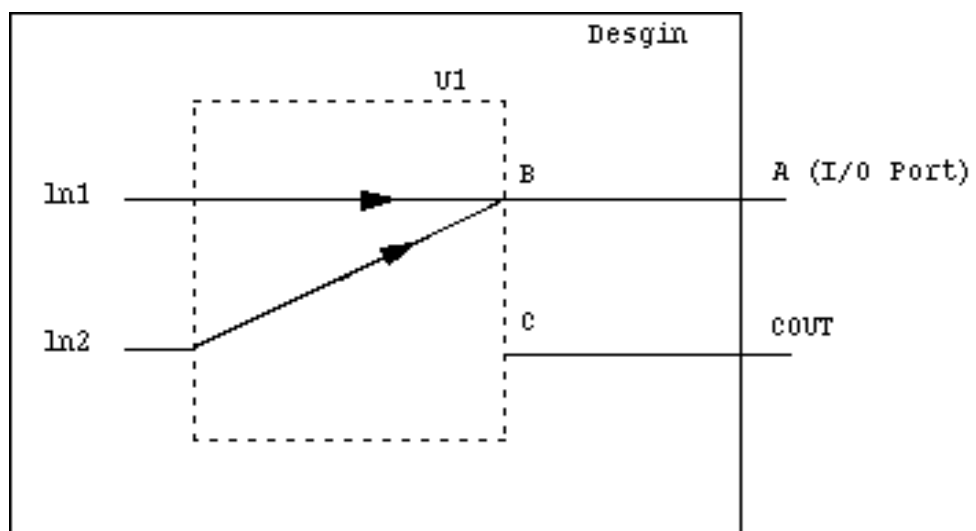
```
DEVICE (1:2:3)
```

VCS assigns the timing value to the generics `tpdIn1_B_R`, `tpdIn2_B_R`, `tpdIn1_C_R`, `tpdIn2_C_R`, `tpdIn1_B_F`, `tpdIn2_B_F`, `tpdIn1_C_F`, and `tpdIn2_C_F`.

Also, it incorrectly assigns timing values to the following generics because port B is an I/O port:

```
tpdB_In1_R, tpdB_In1_F, tpdB_In2_R, and tpdB_In2_F
```

Figure 13-4 Specifying Delays for an I/O Port in a Device



To handle I/O ports, use the INTERCONNECT construct instead of the DEVICE construct.

Using the INTERCONNECT Construct

The INTERCONNECT construct represents the actual or estimated delay in the wire between devices.

The following SDF specifies the loading for Figure 13-4:

```
(INSTANCE CELLTYPE)
(DELAY
(ABSOLUTE
(INTERCONNECT U1/B A (2:3:4))))
```

In the above statements, the interconnect must be reflected at the input of the next stage (which is actually the load or tester in this case). Since this is not available for simulation purposes, this interconnect is added to the output of the previous stage.

But the intent of the SDF entry was to backannotate to all generics that use B as an output port. In this case, the following SDF entry yields correct backannotation:

```
(DEVICE B (1:2:3))
```

Multiple Backannotations to Same Delay Site

VCS backannotates source to load delays onto the delay site associated with the load port. Therefore, there are multiple delay entries, from different source ports to the same load port. In such cases, you can specify the delay value that you want to backannotate.

The SDFPOLICY variable allows you to select the backannotation delay value. You can set this variable to MIN, MAX, or MINOMAX. The default value is MAX for backwards compatibility. If you specify a value other than MIN, MAX, or MINOMAX, VCS issues a warning and uses MAX as the default value.

INTERCONNECT Delays

INTERCONNECT entries in an SDF file are for backannotating delays to a net that connects two or more module instance ports.

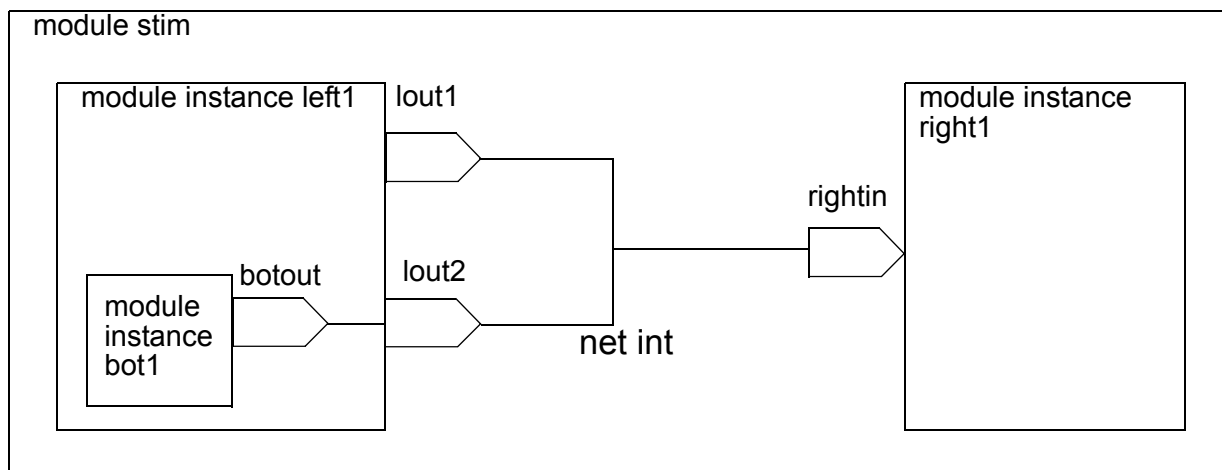
If that net has more than one driver, for example, when a net connects more than one output port to an input port, you can use an SDF file to backannotate different delays between the input port and each of the output ports. Doing so is called backannotating multisource INTERCONNECT delays.

If that net has only one driver, you are backannotating single source INTERCONNECT delays. There is an option for multisource INTERCONNECT delays that can be handy for single source INTERCONNECT delays. See "[Single Source INTERCONNECT Delays](#)" on page 13-36.

Multisource INTERCONNECT Delays

Figure 13-1 shows a multisource net, a net with more than one driver.

Figure 13-1 Net with Multiple Drivers



The SDF file could specify different delays between output port `lout1` and input port `rightin` and output port `botout` and input port `rightin`. The ports do not have to be on the same hierarchical level.

An example for the entries that specify these different delays is as follows:

```
(CELL
  (CELLTYPE "stim")
  (INSTANCE stim)
  (DELAY
    (ABSOLUTE
      (INTERCONNECT stim.left1.bot1.botout stim.right1.rightin (2))
      (INTERCONNECT stim.left1.lout1 stim.right1.rightin(1))
    )
  )
)
```

These entries specify a 2 time unit delay between the output port `botout` and the input port `rightin`, and a 1 time unit delay between output port `lout1` and input port `rightin`.

Note:

In delay value lists in delay definition entries in an SDF file, you can list one, two, three, or six delay values. Lists of twelve values, that also specify delays for transitions to and from X are not yet implemented.

The `+multisource_int_delays` compile-time option tells VCS to use a special algorithm for multisource INTERCONNECT delays. This algorithm simulates different delays from each of the multisource output or inout ports to the load input or inout port, or in other words, true pin to pin delays.

Omitting the `+multisource_int_delays` Option

If you omit the `+multisource_int_delays` option, VCS uses an older algorithm that creates an MIPD (Module Input Port Delay) to model the INTERCONNECT delay. Omitting the option, therefore, has the following drawbacks:

- If you specify multiple sources with multiple INTERCONNECT entries for connecting more than one output or inout port instances to a particular input or inout port instance, this algorithm uses the longest delays in these INTERCONNECT entries for the delays in the MIPD so the delay propagating from all sources will be the same.
- MIPDs only take three delay values for transitions to 1, to 0, and to Z. If your INTERCONNECT entry has six delay values, the MIPD only uses the first three so the Z to 1 delay is the same as the 0 to 1 delay, the 1 to Z delay is the same as the 0 to Z delay, and the Z to 0 delay is the same as the 1 to 0 delay.

Simultaneous Multiple Source Transitions

When there are simultaneous transitions on more than one source port instance, the algorithm for the `+multisource_int_delays` option applies the shortest delay to all of these transitions instead of the one specified in the SDF file.

For example, if the SDF file specifies the following:

```
(CELL
  (CELLTYPE "stim")
  (INSTANCE stim)
  (DELAY
    (ABSOLUTE
      (INTERCONNECT stim.left1.bot1.botout stim.right1.rightin (2))
      (INTERCONNECT stim.left1.lout1 stim.right1.rightin(1))
    )
  )
)
```

Here the delay values are reversed from the previous SDF example. The delay from output port `botout` to `rightin` is 2 and the delay from output port `lout1` to `rightin` is 1.

Figure 13-2 shows the waveforms for these ports and for net `int` that connects these ports and to which you backannotate the INTERCONNECT delay.

Figure 13-2 Simultaneous Source Transitions

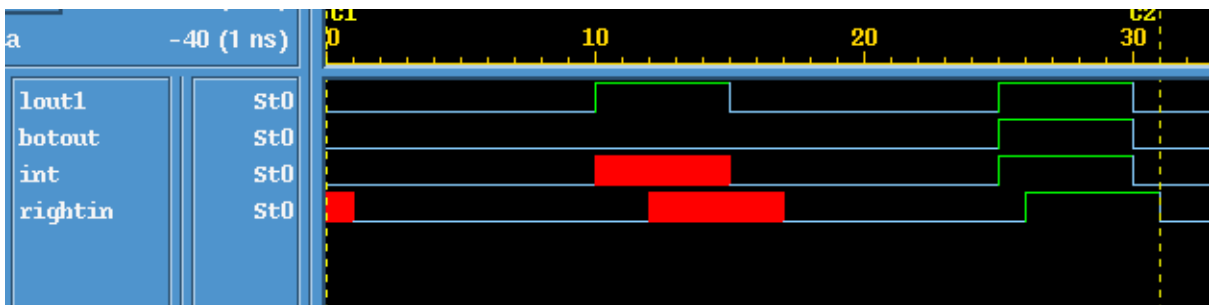


Figure 13-2 illustrates the following series of events:

1. At time 10 output port `lout1` transitions to 1. Net `int` transitions therefore to X and there is a two time unit delay before this X value propagates through `int` to port `rightin`.
2. At time 15 port `lout1` transitions to 0, net `int` transitions to 0 and port `rightin` transitions to 0 two time units later.
3. At time 25 both output ports `lout1` and `botout` transitions to 1. So does net `int`. Input port `rightin` transitions to 1 only one time unit later even though there is a two time unit delay between `lout1` and `rightin`. The algorithm applied the shortest delay, one time unit, to the simultaneous transitions at time 25 on `lout1` and `botout`.

Single Source INTERCONNECT Delays

If the INTERCONNECT entries in your SDF file connect no more than one output or inout port to each of the input or inout ports in your design you should consider including the `+multisource_int_delays` compile-time option that is used for multisource INTERCONNECT delays.

If you omit the `+multisource_int_delays` option, VCS uses an older algorithm that creates a MIPD (Module Input Port Delay) to model the INTERCONNECT delay. MIPDs only take three delay values for transitions to 1, to 0, and to Z. If your INTERCONNECT entry has six delay values, the MIPD only uses the first three so the Z to 1 delay is the same as the 0 to 1 delay, the 1 to Z delay is the same as the 0 to Z delay, and the Z to 0 delay is the same as the 1 to 0 delay.

Note:

In delay value lists in delay definition entries in an SDF file, you can list one, two, three, or six delay values. VCS does not yet support lists of twelve values, that also specify delays for transitions to and from X.

Min:Typ:Max Delays

You can enter min:typ:max delay value triplets wherever you can enter a delay specification. For example:

```
assign #(4:10:14) w1=r1;
```

Here the minimum delay value is 4, the typical delay value is 10, and the maximum delay value is 14.

You can also enter min:typ:max delay value triplets in delay value in entries for different kinds of delays in SDF files. For example, if an SDF file specifies the following:

```
(CELL
  (CELLTYPE "stim")
  (INSTANCE stim)
  (DELAY
    (ABSOLUTE
      (INTERCONNECT stim.left1.lout1
        stim.right1.rightin(6:10:14))
    )
  )
)
```

The INTERCONNECT delay on the net that connects the specified port instances has a minimum delay of 6, a typical delay of 10, and a maximum delay of 14. You can enter min:typ:max delays for delay values in other kinds of delays such as IOPATH or PORT.

Include the `+mindelays` compile-time option to specify using the minimum delay of the `min:typ:max` delay value triplet either in delay specification or in the delay value in entries in an SDF file.

Include the `+maxdelays` compile-time option to specify using the maximum delay.

By default VCS uses the typical delays. You can specify using the typical delays with the `+typdelays` compile-time option.

In the case of SDF files, the `mtm_spec` argument to the `$sdf_annotate` system task overrides the `+mindelays`, `+typdelays`, or `+maxdelays` options.

Specifying Min:Typ:Max Delays at Runtime

If you have either of the following:

- An SDF file that backannotates delays to your design when simulation starts and which contains delay values that are `min:typ:max` delay value triplets
- Module path delays or timing check delays in your Verilog source code are `min:typ:max` delay value triplets

There is a method that enables you to specify using minimum, typical, or maximum delays at runtime instead of at compile-time.

This method is to use the `+allmtm` compile-time option and use the `+mindelays`, `+typdelays`, and `+maxdelays` options at runtime instead of at compile-time.

Using the `+allmtm` compile-time option tells VCS to write three different compiled SDF files when VCS compiles the ASCII text SDF file. One has the minimum delays from the min:typ:max delay value triplets, another has the typical delays, and the third has the maximum delays. You specify which one to backannotate delays from at runtime with the `+mindelays`, `+typdelays`, or `+maxdelays` runtime options.

Using the `+allmtm` compile-time option also tells VCS to prepare the executable so that you can use the `+mindelays`, `+typdelays`, or `+maxdelays` options at runtime to specify using the minimum, typical, or maximum delay values in the min:typ:max delay value triplets in the module path and timing check delays in your source code.

When you use the `+allmtm` compile-time option, the `+typdelays` option is the default at runtime.

This method does not apply to other delay specifications in your source code that might contain min:typ:max delay value triplets, such as gate delays or delays in continuous assignments.

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 3-37](#) 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. Timopt 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 RadiantTechnology (`+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 it's not sure of.
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. Now 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. Timopt makes its optimizations based on its analysis of the design and information in the timopt.cfg file. If during simulation 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 like 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 {tfnpc} {,};
```

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 100 ns 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 case 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 case 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 from in front of the signal's hierarchical name.

14

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 chapter describes their purpose, how they work, and how to use them in the following sections:

- [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 Backannotation to Negative Timing Checks](#)

- [How VCS Calculates Delays](#)
- [Using Multiple Non-Overlapping Violation Windows](#)

The Need for Negative Value Timing Checks

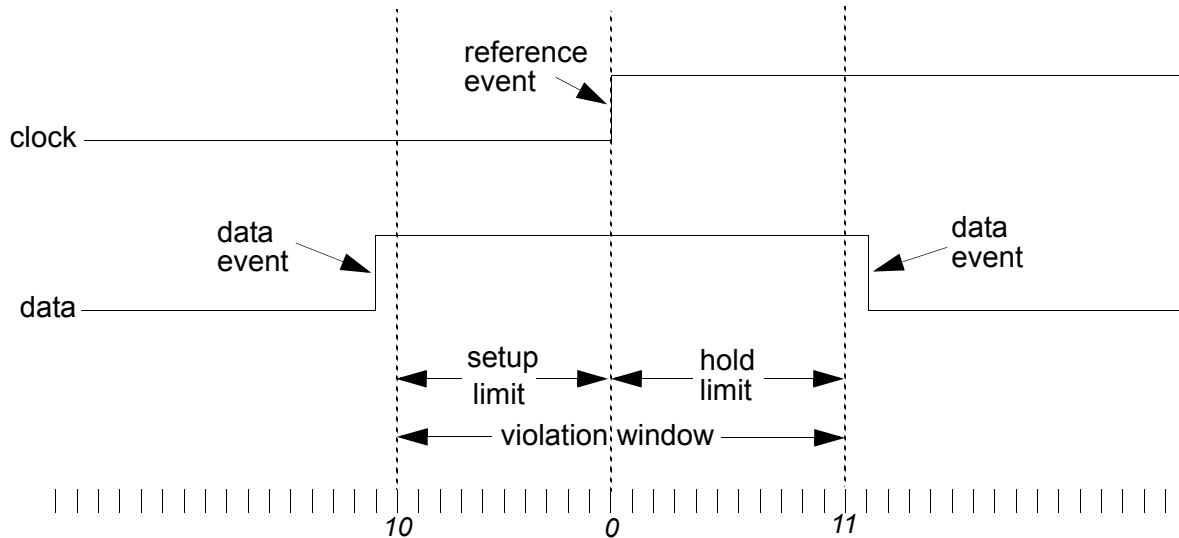
Negative Timing Checks for XYZ

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 like 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);
```

Here 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 notifier register, in this example `reg notifyreg`. You could use this toggling of a notifier register to output an X value from a device, such as a sequential flop, when there is a timing violation.

Figure 14-1 Positive Setup and Hold Limits



Here both the setup and hold limits have positive values. When this happens the violation window straddles the reference event.

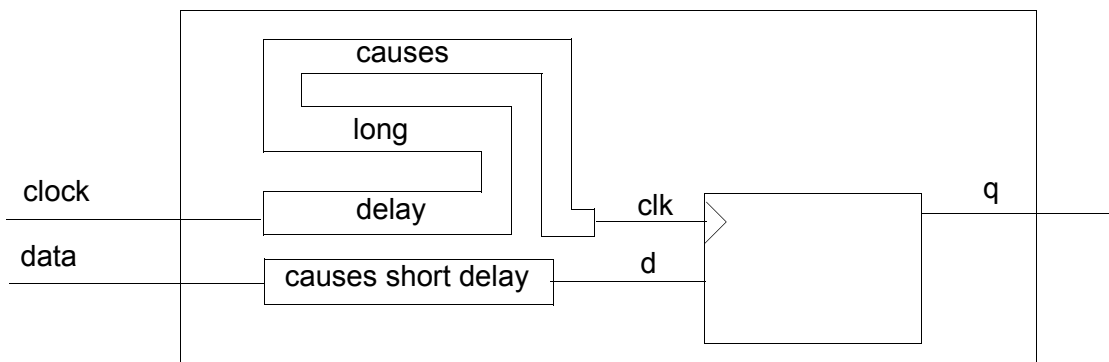
There are cases where the violation window cannot straddle the reference event at the inputs of an ASIC cell. Such a case occurs when:

- 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 14-2 ASIC Cell with Long Propagation Delays on Reference Events

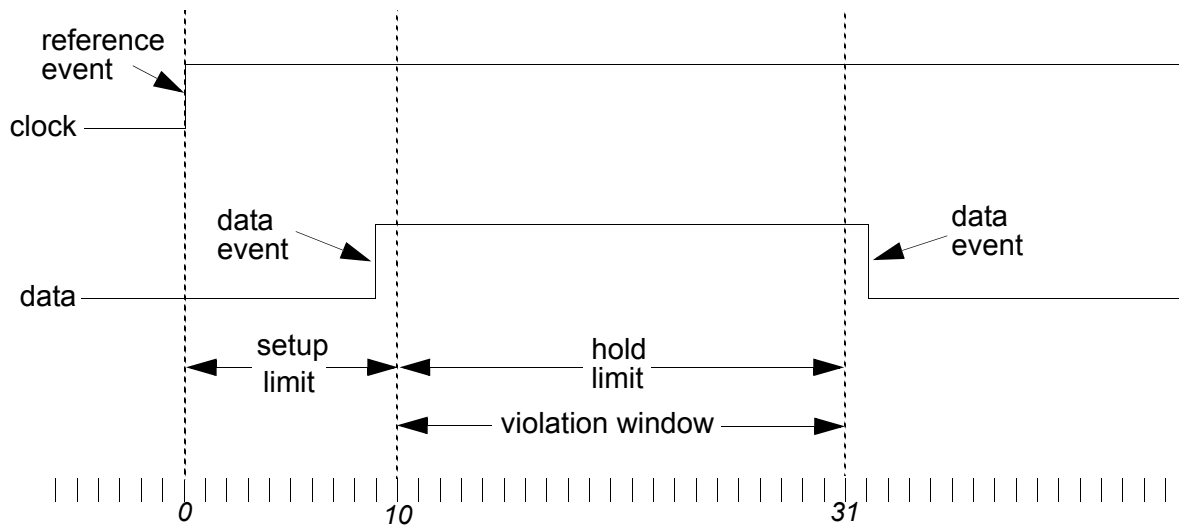


When this happens the violation window shifts at the cell boundary so that it no longer straddles the reference event. It shifts to the right when there are longer propagation delays on the reference event. This right shift requires a negative setup limit:

```
$setuphold (posedge clock, data, -10, 31, notifyreg);
```

Figure 14-3 illustrates this scenario.

Figure 14-3 Negative Setup Limit



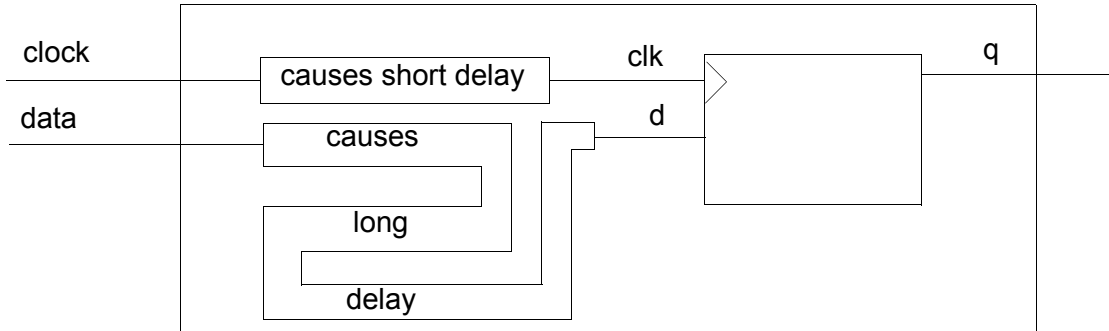
Here the `$setuphold` timing check is in the specify block of the top-level module of the cell. It specifies that there is a timing violation if there is a data event between 10 and 31 time units after the reference event on the cell boundary.

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 14-4 ASIC Cell with Long Propagation Delays on Data Events

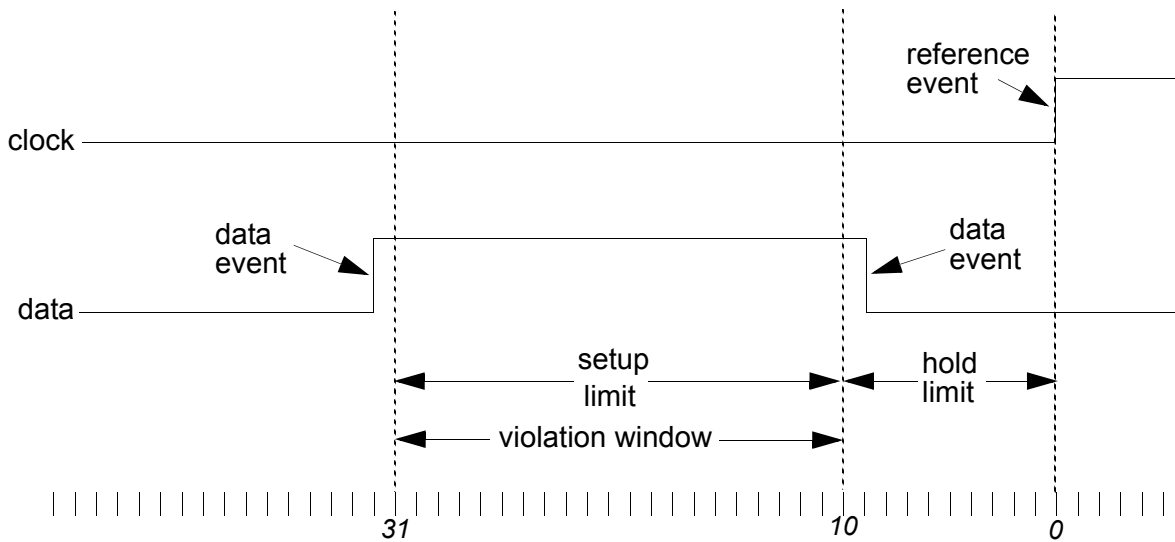


The violation window shifts to the left when there are longer propagation delays on the data event. This left shift requires a negative hold limit:

```
$setuphold (posedge clock, data, 31, -10, notifyreg);
```

Figure 14-5 illustrates this scenario.

Figure 14-5 Negative Hold Limit



Here 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 using the extended syntax of the `$setuphold` system task or let VCS 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 `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 of these events is. 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 `$recrem` timing check is for checking how close asynchronous control signal transitions are to clock signals. Like the setup and hold limits in `$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 as a reference signal like a clock signal and data signal can have different propagation delays from the cell boundary to a sequential device inside the cell, there 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, like 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 `$recrem` 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 `$recrem` 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" 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 `$setuphold` and `$recrem` 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 14-1 illustrates this:

Example 14-1 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:

```

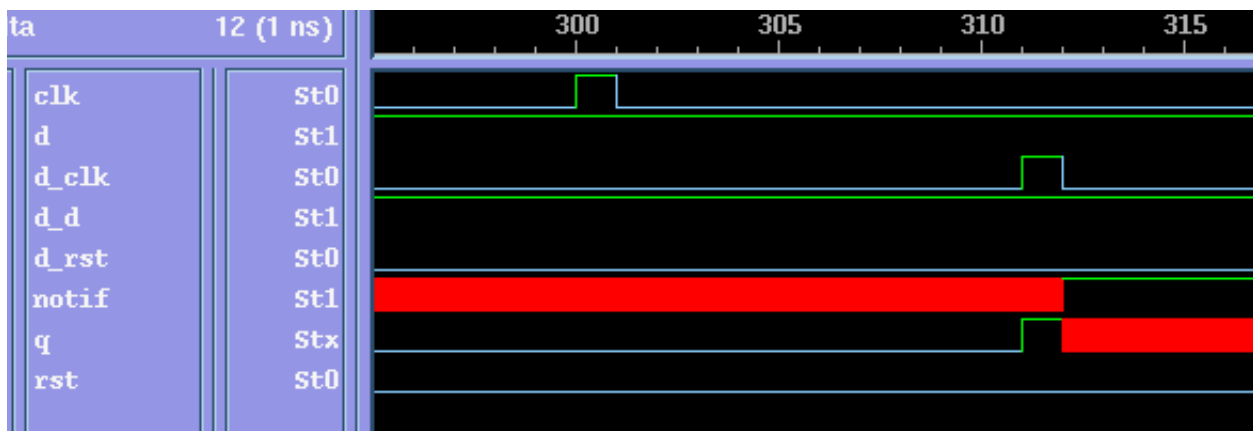
"test1.v", 31: Timing violation in top.dff1
$width( posedge clk:300,      : 301, limit: 5);

```

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 14-2 Other Timing Checks Using the Delayed Versions

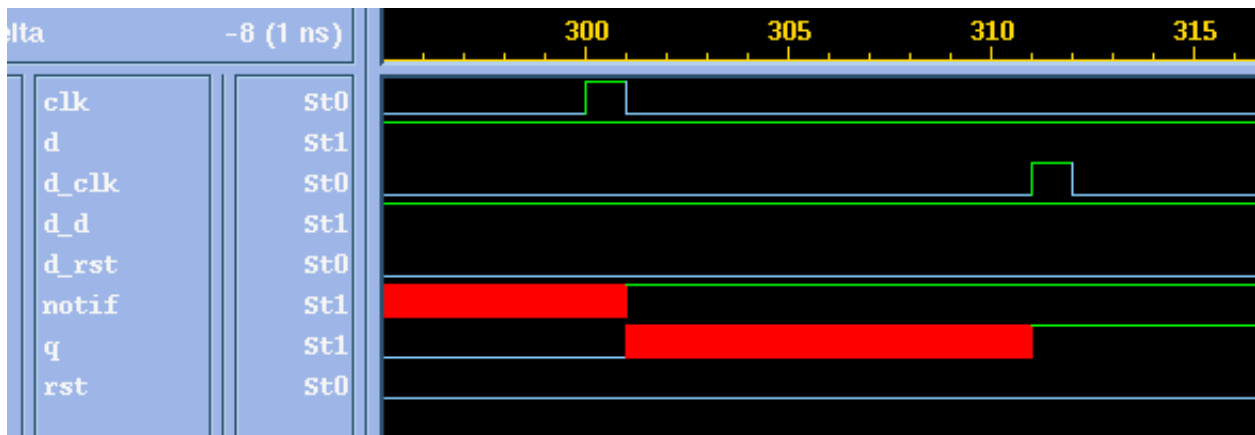


If you include the `+neg_tchk` compile-time option and also the `+old_ntc` compile-time option, the `$width` timing check does not use the delayed version of signal `clk` and the following sequence of events occurs:

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. This in turn 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` changing the output `q` to 1 erasing the X value on this output.

Figure 14-3 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 or 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 Backannotation to Negative Timing Checks

You can backannotate 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 backannotate 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 backannotation to occur.
- If there is no `SCOND` or `CCOND` expressions in the SDF statement, all timing checks that otherwise match are backannotated.

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:

$$20 > (\text{delay on del_clock} - \text{delay on del_data}) > 10$$

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);
```

Here:

- 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. That 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 backannotated from an SDF file.
- The violation windows do not converge or overlap.

The default behavior of VCS when these conditions are met 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;
end
```

```

        #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 backannotate 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))
  )
)

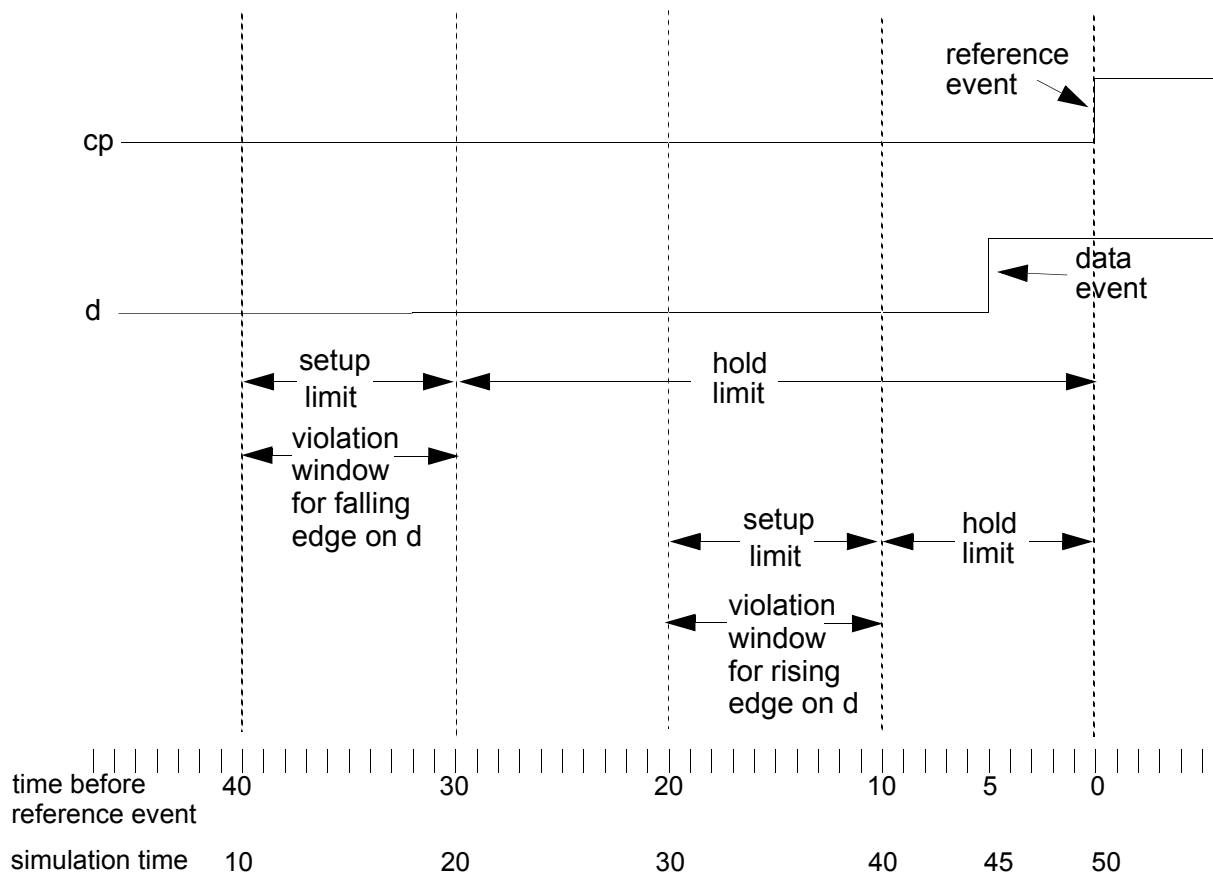
```


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.fdl_1.cp` and `top.fdl_1.d` are shown in Figure 14-4.

Figure 14-4 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 what you want 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, the reference event, VCS displays the timing violation:

```
"sourcefile.v", line_number_of_second_timing_check:  
Timing violation in top.fdl_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.

Negative Timing Checks

14-28

15

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](#)
- [Typical Flow to Dump the Backward SAIF File using System Tasks](#)
- [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 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]);
```

`$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.

Typical Flow to Dump the Backward SAIF File using System Tasks

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.

16

SWIFT VMC Models and SmartModels

VMC models are secure, protected, and portable models of Verilog designs. They contain no Verilog code. SmartModels are models from Synopsys that model devices from various vendors. SWIFT is the interface for both of these kinds of models. VCS enables you to instantiate both these kinds of models in your design and simulate them as part of your design. The steps you take are as follows:

1. Set the SWIFT environment variables.
2. Generate a Verilog template file for the model. You use this template file to instantiate the model.
3. Enable the monitoring of signals inside the model through the model window.
4. Enter commands for the model in your source code using the LMTV window commands or the SWIFT command channel.

5. Compile the design with compile-time options for the SWIFT interface and then simulate your design.

This chapter describes these steps in the following topics:

- [SWIFT Environment Variables](#)
- [Generating Verilog Templates](#)
- [Monitoring Signals in the Model Window](#)
- [Using LMTV SmartModel Window Commands](#)
- [Entering Commands Using the SWIFT Command Channel](#)
- [Loading Memories at the Start of Runtime](#)
- [Compiling and Simulating a Model](#)

Note:

The information in this chapter is provided as a convenience to the Synopsys model user. It includes basic information about simulating Synopsys models. This chapter is not, however, the authoritative source on this subject. More complete and sometimes more up-to-date information can be found in the *Simulator Configuration Guide for Synopsys Models*. This guide is available on-line in PDF format at <http://www.synopsys.com/products/lm/doc/smartmodel/manuals/simcfg.pdf> or <http://www.synopsys.com/products/lm/doc/hardwaremodel/manuals/simcfg.pdf>

SWIFT Environment Variables

You set some SWIFT environment variables on all platforms. Others you set only on certain platforms.

All Platforms

You must set the LMC_HOME environment variable to the directory where you installed the models. For example:

```
setenv LMC_HOME /u/tools/swiftR41
```

Set the VCS_SWIFT_NOTES environment variable to 1. For example:

```
setenv VCS_SWIFT_NOTES 1
```

Setting this environment variable enables the display of messages from models including messages that tell you if the model is correctly loaded. For example, if PCL code contains printf commands, during the simulation the microprocessor controlled by the PCL code prints out the model messages.

Setting this environment variable is optional but Synopsys recommends that you always set it when you are debugging your design.

Solaris Platform

For the Solaris platform, set the environment variable LD_LIBRARY_PATH. If you have already set this environment variable for some other application, you can add to the definition of this environment variable as follows:

```
setenv LD_LIBRARY_PATH ${LMC_HOME}/lib/sun4Solaris.lib:  
${LD_LIBRARY_PATH}
```

If you haven't already set this environment variable, enter the following:

```
setenv LD_LIBRARY_PATH ${LMC_HOME}/lib/sun4Solaris.lib
```

HP Platform

For the HP platform, set the SHLIB_PATH environment variable. If you have already set this environment variable for some other application, you can add to the definition of this environment variable as follows:

```
setenv SHLIB_PATH ${LMC_HOME}/lib/hp700.lib:${SHLIB_PATH}
```

If you haven't already set this environment variable, enter the following

```
setenv SHLIB_PATH ${LMC_HOME}/lib/hp700.lib
```

Linux

For the Linux platform, set the LD_LIBRARY_PATH environment variable as follows:

```
% setenv LD_LIBRARY_PATH $LMC_HOME/lib/  
x86_linux.lib:$LD_LIBRARY_PATH
```

Generating Verilog Templates

You can generate the Verilog template for the model using the `-lmc-swift-template` option. The `vcs` command line to do this is as follows:

```
vcs -lmc-swift-template modelname
```


This command generates a Verilog template file named *modelname.swift.v*. For example, if you enter the following `vcs` command:

```
vcs -lmc-swift-template xc4062xl_432
```

VCS writes the `xc4062xl_432.swift.v` file in the current directory.

The Verilog template file contains a Verilog module definition that contains:

- The special `$vcs_swift` user-defined system task for the SWIFT interface that enables you to use the command channel to the SWIFT interface to pass commands to the model and see messages from the model.
- Declarations for window regs that enable you to see the value of and, in some cases, deposit values to signals in the model. See ["Monitoring Signals in the Model Window" on page 16-8](#).
- Declarations for regs that you use to pass commands to the model.
- Port and reg declarations and assignment statements that are part of a Verilog shell for the model.

When you instantiate the module definition in this Verilog template file, you instantiate the model.

Modifying the Verilog Template File

You can make certain modifications to the contents of the Verilog template file.

The modifications you can make are as follows:

Reordering ports in the module header

If, for example, the module header is as follows:

```
module xyz (IO0, IO1, IO2, IO3, IO4);
```

You can reorder the ports:

```
module xyz (IO4, IO3, IO2, IO1, IO0);
```

Concatenating ports in the module header

You can concatenate ports in the module header, for example:

```
module xyz ({IO4, IO3, IO2, IO1}, IO0);
```

Doing so enables you to connect vector signals to the model as follows:

```
wire [3:0] bus4;  
...  
xyz xyz1( bus4, ...
```

Naming concatenation expressions in the module header

In Verilog you can name concatenation expressions in the port connection list in a module header. For example:

```
module xyz (.IO({IO4, IO3, IO2, IO1}), IO0);
```

This allows you to use name based connections in the module instantiation statement, as follows:

```
wire [3:0] bus4;  
..  
xyz xyz1( sig1, .IO(bus4), ...
```

Redefining Parameters

The Verilog template file contains a number of parameters that are used for specifying model attributes. In some cases you can modify the parameter definition. For example, the template file contains the following parameter definition:

```
parameter DelayRange = "MAX";
```

This parameter specifies using the maximum delays of min:typ:max delay triplets in the model. You can change the definition to "TYP" or "MIN". There is an alternative to editing the `DelayRange` parameter definition; see ["Changing the Timing of a Model" on page 16-16](#).

For another example, the template file for a memory model might contain the following parameter definition:

```
parameter MemoryFile = "memory";
```

If you know that all instances of this model will need to load memory file `mem.dat`, you can change this to:

```
parameter MemoryFile = "mem.dat";
```

You can also use `defparam` statements to change these parameter definitions. For example, if an instance of a memory model has an instance name of `test.design.mem1` and this model must load memory file `mem1.dat`, you can enter the following in the test fixture module:

```
defparam design.mem1.MemoryFile = "mem1.dat";
```

For more information on SmartModel attributes that are parameters in the Verilog template file, see the *SmartModel Library Simulator Interface Manual*.

Monitoring Signals in the Model Window

SWIFT VMC models and SmartModels can have a window that enables you to see the values of certain signals inside the model. For some models you can also deposit values on these signals. The model-specific data sheet lists which signals that you can monitor in the window and whether you can also deposit values to these signals.

When you generate the Verilog template file for a model, VCS declares window regs in the template file that correspond to these signals inside the model window. By monitoring the window regs, you monitor the corresponding signals in the model. Assigning values to these window regs deposits values to the corresponding signals in the model.

To enable VCS to declare these regs for SmartCircuit models, do the following:

1. Create the file listing the SmartCircuit windows. Refer to the *SmartModel Library Users Manual* for a description of how to do this.
2. Create a soft link or copy the Model Control File (MCF) to a file named scf in the current directory. VCS uses this file to load the netlist for the SmartCircuit model that contains the signals in the window.

The following are examples of these regs in the Verilog template file:

```
//Generating SmartModel Windows data
reg [1:0] GEN_CCLK_SPEED;
reg [4:0] LENGTH_CNT_WIDTH;
reg [10:0] FRAME_SIZE;
reg [12:0] DEVICE_FRAMES;
reg [3:0] CRC_ERROR_CHK;
reg SYNC_TO_DONE;
reg [3:0] DONE_ACTIVE;
reg [3:0] IO_ACTIVE;
reg [3:0] DEVICE_STATE;
reg CONFIGURATIONMODE;
```

You enable and disable the monitoring of these window regs with the special `$swift_window_monitor_on` and `$swift_window_monitor_off` system tasks. The syntax for these system tasks are as follows:

```
$swift_window_monitor_on("instance_name" [, "window_reg",
"window_reg", ...]);
$swift_window_monitor_off("instance_name" [, "window_reg",
"window_reg", ...]);
```

Here:

<i>instance_name</i>	Specifies the hierarchical name of the instance of the module definition in the Verilog template file.
<i>window_reg</i>	The identifier of the window reg in the Verilog template file. If you do not specify a window reg in these system tasks, you enable or disable the monitoring of all the window regs in the instance.

The following are examples of using these system tasks:

Example 1

```
$swift_window_monitor_on("test.design.xc4062x1_432_1");
```

This example enables the monitoring of all window regs in the module instance for the model with the hierarchical name `test.design.xc4062xl_432_1`.

Example 2

```
$swift_window_monitor_on("test.design.xc4062xl_432_1",  
"GEN_CCLK_SPEED", "LENGTH_CNT_WIDTH");
```

This example enables the monitoring of the window regs `GEN_CCLK_SPEED` and `LENGTH_CNT_WIDTH` in the module instance `test.design.xc4062xl_432_1`.

After you enable monitoring or depositing with these system tasks you must specify the monitoring with, for example, the `$monitor` system task, and depositing values with procedural assignments to these window regs.

Using LMTV SmartModel Window Commands

VCS supports a number of the Logic Model to Verilog (LMTV) SmartModel window commands that were implemented as a command interface between SmartModels and the previous generation of Verilog simulators. These commands are user-defined system tasks that we provide for simulating with SmartModels. They also work with VMC models.

These system tasks are as follows:

```
$lm_monitor_enable
```

Enables SmartModel windows for one or more window elements in a specified model instance. This system task is functionally

equivalent to the `$swift_window_monitor_on` system task described in ["Monitoring Signals in the Model Window" on page 16-8](#). Its syntax is as follows:

```
$lm_monitor_enable(regname,instance_name,  
"window_element")
```

`$lm_monitor_disable`

Disables SmartModel windows for one or more window elements in a specified model instance. This system task is functionally equivalent to the `$swift_window_monitor_off` system task described in ["Monitoring Signals in the Model Window" on page 16-8](#). Its syntax is as follows:

```
$lm_monitor_disable(regname,instance_name,  
"window_element")
```

`$lm_command`

Sends a command to the session or to a model instance. You use this system task to pass Model and Session commands to a SmartModel, SmartBrowser commands to a SmartCircuit model, and PCL commands to a microprocessor model. Its syntax is as follows:

```
$lm_command ("session_cmmd_string" |  
"instance_name", "model_cmmd_string");
```

`$lm_dump_file`

Dumps the memory contents of the specified instance into the specified file. This works only for memory models. Overwrites the specified file if it already exists. Using this system task eliminates the read cycles required to verify the success of a test. Its syntax is as follows:

```
$lm_dump_file("instance_name", "filename"  
[, "MEMORY"]);
```

`$lm_load_file`

Loads the memory contents of a file into an instance (the contents of this file are in a specific memory dump format). The instance can be either a programmable device or a memory model. Using this system task eliminates the write cycles required to set up the contents of the model. You can also use this system task to load a model control file (MCF) during simulation. Its syntax is as follows:

```
$lm_load_file("instance_name", "filename"  
[, "MEMORY|JEDEC|PCL|SCF|MCF"]);
```

Model, Session, PCL, and SmartBrowser commands are described in the *SmartModel Library User's Manual*.

Note:

The *instance_name* argument in the `$lm_monitor_enable` and `$lm_monitor_disable` system tasks is not enclosed in quotation marks whereas the quotation marks are required in the `$lm_command`, `$lm_dump_file`, and `$lm_load_file` system tasks.

For complete and authoritative information on these LMTV SmartModel window commands, see the *SmartModel Library Simulator Interface Manual*. The information on these commands is in the interface information for Verilog-XL but this information is also good for VCS. VCS supports these tasks to make it easy for people to move from Verilog-XL to VCS.

You can access these documents at <http://www.synopsys.com/products/lm/docs>.

Note:

VCS does not support two-dimensional windows for memory models and therefore has not implemented other LMTV window commands.

Entering Commands Using the SWIFT Command Channel

As an alternative to using the LMTV window commands, you can use the SWIFT command channel to pass commands to a SmartModel or a VMC model.

The command channel works by assigning these commands and toggling the values of command channel regs declared in the Verilog template file. The regs to which you assign these values are as follows:

Reg	Description
<code>cmd\$str</code>	The reg to which you assign the command
<code>do\$model\$cmd</code>	When you assign a value of 1 to this reg, the model executes all Model, SmartBrowser, and LMTV commands assigned to reg <code>cmd\$str</code> .
<code>do\$session\$cmd</code>	When you assign a value of 1 to this reg, the model executes all Session commands assigned to reg <code>cmd\$str</code> .
<code>log\$file</code>	The reg to which you assign a logfile name. The model writes to this logfile when <code>log\$on</code> has a value of 1.
<code>log\$on</code>	When you assign a value of 1 to this reg you enable the model to write to the logfile you assigned to reg <code>log\$file</code> .

The following is an example of an initial block that passes commands through the command channel:

```
initial
begin
#1 circuit.model.cmd$str = "show doc";
circuit.model.do$model$cmd=1 ; // 1
#1 circuit.model.do$model$cmd=0 ;
#1 circuit.model.cmd$str = "show timing unit";
circuit.model.do$model$cmd = 1;
#1 circuit.model.do$model$cmd=0 ;
#1 circuit.model.cmd$str = "show version";
circuit.model.do$model$cmd = 1;
#1;
end
```

The Verilog template files for SmartModel memory models also contain the following reg declarations that allow you to write or dump the contents of a memory to a file:

<code>mem\$dump\$file</code>	The reg to which you assign the name of the file into which the memory model writes its contents. The model writes to this file when <code>do\$mem\$dump</code> has a value of 1.
<code>do\$mem\$dump</code>	Enables writing the memory models contents to the file.

For example, if you had a memory model with the hierarchical name `top.asic.mem1` and you wanted to dump its contents to file `mem1.1k.dump` at time 1000, and `mem1.2k.dump` at time 2000, you could use the following initial block in your test fixture module:

```
initial
begin
#1000 top.asic.mem1.mem$dump$file = "mem1.1k.dump";
top.asic.mem1.do$mem$dump = 1;
#1000 top.asic.mem1.mem$dump$file = "mem1.2k.dump";
top.asic.mem1.do$mem$dump = 1;
end
```

Using the CLI to Access the Command Channel

You can also use the CLI to access the command channel during simulation. For example:

```
cli_0> set circuit.model.cmd$str = "show doc";
cli_1> once #1;
cli_2> .
cli_3> set circuit.model.do$model$cmd=1;
cli_4> once #1;
cli_5> .
```

Loading Memories at the Start of Runtime

SmartModel memory models have an attribute called `MemoryFile`. You can use this attribute to load the contents of a file into these memories at runtime. To do so use a `defparam` statement. For example:

```
defparam
test.designinst.modelinst.MemoryFile="mem_vec_file";
```

Here VCS treats the attribute as if it were a parameter in an instance.

This method enables you to load a memory when simulation starts. To load a memory after simulation starts use the LMTV window command `$lm_load_file` system task. Refer to ["Using LMTV SmartModel Window Commands" on page 16-10](#).

Compiling and Simulating a Model

If your design instantiates a SmartModel or a VMC model, you compile your design with the `-lmc-swift` compile-time option. Be sure to also include the Verilog template file on the `vcs` command line. For example:

```
vcs -lmc-swift xc4062x1_432.swift.v test.v design.v
```

This command line results in an executable file named `simv`. Enter this executable file on a command line to simulate the design that instantiates the model:

```
simv
```

Changing the Timing of a Model

You can enter the `+override_model_delays` runtime option in combination with either the `+mindelays`, `+typdelays`, or `+maxdelays` option to override the `DelayRange` parameter in the template file that specifies the timing used by the model.

If you use this method, all the SmartModel models in your design will use either the minimum, typical, or maximum delays specified by the `+mindelays`, `+typdelays`, or `+maxdelays` option.

If you want to use different timing options in different models in your design you must edit the template files for each model to change the `DelayRange` parameter definition to either "MIN", "TYP", or "MAX".

17

Using the 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 has implemented the TF and ACC routines for the PLI. It has also implemented the VPI procedural interface routine to some extent.

VCS also supports a number of ACC routines that are not part of the IEEE Verilog language reference manual. These routines access:

- Reading and writing to memories
- Multi-dimensional arrays

- Probabilistic distribution
- Returning a string pointer to a parameter value
- Extended VCD files
- Line callbacks
- Source protection
- Signal in a generate block

ACC routines, which access the values in a design, changing, for example, the delay values of module paths or the simulation values of individual signals, are more powerful than the TF routines which operate only on data passed to them. However, the ACC routines also have a greater performance cost. The ability of ACC routines to traverse the entire design and make extensive value changes requires VCS to omit powerful performance optimizations so that it is possible for the ACC routines to make these changes.

This performance cost of the ACC routines is a major consideration in VCS performance. There are ways to limit this performance cost and doing so is an important step in developing a PLI application that uses ACC routines.

This chapter covers the following topics:

- [Writing a PLI Application](#)
- [Functions in a PLI Application](#)
- [Header Files for PLI Applications](#)
- [The PLI Table File](#)
- [Enabling ACC Capabilities](#)

- [Using VPI Routines](#)
- [Writing Your Own main\(\) Routine](#)

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 TF and ACC routines that Synopsys has implemented 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 "[The PLI Table File](#)" on page 17-6). 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, do 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 of the ACC routines called by your functions for faster performance.
- Limits scope and operations of the ACC routines 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's 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 `$stop`, or `$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 functions are the ones 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:

`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.

You will find these header files at `$VCS_HOME/platform/lib`, where *platform* is the platform you are using, such as `sun_sparc_solaris_5.7`.

The header file for the VPI routines is `$VCS_HOME/include/vpi_user.h`; see ["Using VPI Routines" on page 17-29](#).

The PLI Table File

The PLI table file (commonly called the `pli.tab` file) has two purposes:

- To associate user-defined system tasks and system functions with functions in a PLI application, so VCS calls these functions when it compiles or executes the system task or function.
- To limit the scope and operation of the ACC routines called by the debugging features. If this is all that you need to do in the PLI table file, see ["Specifying ACC Capabilities for PLI Functions" on page 17-12](#) and ["Specifying ACC Capabilities for VCS Debugging Features" on page 17-17](#).

When writing a PLI application, after you write the functions for the application, you need to write a PLI table file. This file contains a line for each user-defined system task or system function your application needs. Each line should contain the following information:

- The name of the user-defined system task or system function. This name begins with the dollar sign `$`.

- The PLI specifications for the user-defined system task or system function. These specifications must include the call function. For user-defined system functions, they must also include the size of the return value.

These specifications can also include the check and miscellaneous functions. There are a number of other PLI specifications, such as the one that allows the entering of the user-defined system task on the command line. You can also enter these specifications on a line in the PLI table file.

- The ACC capabilities that you want to enable for the functions, particularly the call function, for the user-defined system task or system function. When you specify ACC capabilities you specify the types of ACC operations that the ACC routines will perform and where in the design they can perform these operations. Entering ACC capabilities to limit ACC routine scope and operations is optional but recommended to enhance performance.

The syntax for a line in a PLI table file is as follows:

```
$name PLI_specifications [ACC_capabilities]
```

Here:

\$name

The name of the user-defined system task or system function

PLI_specifications

One or more specifications such as the name of the C function VCS calls when it executes the user-defined system task or system function. For a list of valid PLI specifications, see ["PLI Specifications" on page 17-9](#).

ACC_capabilities

Specifications for ACC capabilities to be added, removed, or changed in various parts of the design hierarchy. For details on ACC capabilities, see ["ACC Capabilities" on page 17-11](#).

PLI Specifications

The valid PLI specifications are as follows:

call=function

Specifies the name of call function. This specification is required.

check=function

Specifies the name of check function.

misc=function

Specifies the name of misc function.

data=integer

Specifies the data value passed as first argument to call, check, and misc routines. The default is 0. Use this argument if you want more than one user-defined system task or system function to use the same call, check, or misc function. Specify a different integer for each user-defined system task or system function in the PLI table file that uses the same call, check, or misc function.

size=number

Specifies the size of returned value in bits. This specification is required for user-defined system functions. You can omit this specification or specify 0 in the PLI specifications for a user-defined system function. For user-defined system functions, specify a decimal value for the number or 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 to the user-defined system task or system function.

minargs=number

Specifies the minimum number or arguments.

`maxargs=number`

Specifies the maximum number of arguments.

`nocelldefinepli`

Disables the dumping of value change and simulation time data, from modules defined under the `\celldefine` compiler directive, into the VPD file created by the `$vcdpluson` system task. You make this change in the line for the `$xvcs` system task in the `virsim.tab`, `virsim_dki.tab`, and `virsim_pp.tab` files in the `$VCS_HOME/virsim_support/vcdplus` directory. It doesn't make sense for you to use this specification in a PLI table file for any other PLI application.

This capability is intended for batch simulation only.

`persistent`

Enables you to enter the user-defined system task on the CLI or DVE command line without including any of the `+cli` compile-time options.

The following are example lines in PLI table files that contain PLI specifications (the ACC capabilities parts are omitted):

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

```
$value_passer size=0 args=2 call=value_passer persistent
```

In this line, there is an associated user-defined system task (because it has a return size of 0). The system task takes two arguments. When VCS executes the `$value_passer` system task it calls the function named `value_passer`, and you can enter the system task on the CLI command line without including the `+cli` compile-time option.

Example 3

```
$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
```

ACC Capabilities

You can specify ACC capabilities in a PLI table file for the following reasons:

- To specify the ACC capabilities for the PLI functions associated with your user-defined system task or system function. To do this, specify the ACC 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 ACC Capabilities for PLI Functions" on page 17-12](#) for more details.

- To specify the ACC capabilities that the debugging features of VCS can use. To do this, specify ACC capabilities alone on a line in a PLI table file, without an associated user-defined system task or system function. See ["Specifying ACC Capabilities for VCS Debugging Features"](#) on page 17-17 for more details.

When you specify ACC capabilities, you specify both of the following:

- The ACC capabilities you want to enable or disable.
- The part or parts of the design that you want this enabling or disabling to occur.

In many ways, specifying ACC capabilities for your PLI functions, and specifying them for VCS debugging features, is the same, but the capabilities that you enable, and the parts of the design to which you can apply them are different.

Specifying ACC Capabilities for PLI Functions

The format for specifying ACC capabilities is as follows:

```
acc=|+=|-=|:=capabilities:module_names[+]|%CELL|%TASK|*
```

Here:

`acc`

Keyword that begins a line for specifying ACC capabilities.

`=|+=|-=|:=`

Operators for adding, removing, or changing ACC capabilities.

The operators in this syntax are as follows:

`=`

A shorthand for `+=`.

`+=`

Specifies adding the ACC capabilities that follow to the parts of the design that follow, as specified by module name, `%CELL`, `%TASK`, or `*` wildcard character.

`-=`

Specifies removing the ACC capabilities that follow from the parts of the design that follow, as specified by module name, `%CELL`, `%TASK`, or `*` wildcard character.

`:=`

Specifies changing the ACC 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 ACC capabilities. The ACC 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`

Backannotates timing check delay values.

`gate` **or** `gate_backannotation`

Backannotates delay values on gates.

`mp` **or** `module_path_backannotation`

Backannotates module path delays.

`mip` **or** `module_input_port_backannotation`

Backannotates delays on module input ports.

`mipb` **or** `module_input_port_bit_backannotation`

Backannotates 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 ACC capability in all instances of the specified module.

+

Specifies adding, removing, or changing the ACC 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 ACC 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` compile-time options).

%TASK

Enables, disables, or changes (depending on the operator) the ability of the PLI function to use the ACC 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 ACC capability throughout the entire design. Using wildcard characters could seriously impede the performance of VCS.

Note:

There are no blank spaces when specifying ACC capabilities.

The following examples are the PLI specification examples from the previous section with ACC 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 ACC capabilities for reading and writing to nets and registers, and for backannotating 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 ACC capability for backannotating 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 ACC 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 ACC 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 ACC 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 ACC Capabilities for VCS Debugging Features

The format for specifying ACC capabilities for VCS debugging features is as follows:

```
acc=|+=|-=|:=capabilities:module_names[+]|%CELL|*
```

Here:

acc

Keyword that begins a line for specifying ACC capabilities.

=|+=|-=|:=

Operators for adding, removing, or changing ACC capabilities.

capabilities

Comma separated list of ACC capabilities.

module_names

Comma-separated list of module identifiers. The specified ACC capabilities will be added, removed, or changed for all instances of these modules.

+

Specifies adding, removing, or changing the ACC 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` compile-time options.)

*

Specifies all modules in the design. Using a wildcard character is no more efficient than using the `+cli` compile-time option.

The ACC 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 doing the following (the actual CLI commands described are in parentheses):</p> <ul style="list-style-type: none">• Creating an alias for another CLI command (<code>alias</code>)• Displaying CLI help (<code>?</code> and <code>help</code>)• Specifying the radix of displayed simulation values (<code>offormat</code>)• Displaying simulation values (<code>print</code>)• Descending and ascending the module hierarchy (<code>scope</code>)• Depositing values on registers (<code>set</code>)• Displaying the set breakpoints on signals (<code>show break</code>)• Displaying the port names of the current location, and the current module instance or scope, in the module hierarchy (<code>show ports</code>)• Displaying the names of instances in the current module instance or scope (<code>show scopes</code>)• Displaying the nets and registers in the current scope (<code>show variables</code>)• Moving up the module hierarchy (<code>upscope</code>)• Deleting an alias for another CLI command (<code>unalias</code>)• Ending the simulation (<code>finish</code>)
<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`

Commands for doing the following (the actual CLI commands described are in parentheses):

- Setting a repeating breakpoint. In other words always halting simulation, when a specified signal changes value (`always` or `break`)
- 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 (`once` or `tbreak`)
- Removing a breakpoint from a signal (`delete`)
- Showing the line number or number in the source code of the statement or statements that causes the current value of a net (`show drivers`)

A longer way to specify this capability is with the `callback` keyword.

`frc` or `force`

Commands for doing the following, (the actual CLI commands described are in parentheses):

- Forcing a net or a register to a specified value so that this value cannot be changed by subsequent simulation events in the design (`force`)
- Releasing a net or register from its forced value (`release`)

A longer way to specify this capability is with the `force` keyword.

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
```

Here 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, or 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 ["The PLI Table File" on page 17-6](#)), 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 Enabling ACC Capabilities

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 ["The PLI Table File" on page 17-6](#).

The *level_number* in this option specifies more and more 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.

Enabling ACC Write Capabilities 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.

`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: ()

Using Only the ACC Capabilities that You Need

There are compile-time and runtime options that enable VCS and PLI applications to use only the ACC capabilities they need and no more than they need. 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`, that you can use to recompile your design so that subsequent simulations only use the ACC capabilities that they need.
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 ACC Capabilities are Used

You include the `+vcs+learn+pli` runtime option to tell VCS to learn the ACC capabilities that were used by the modules in your design and write them into a secondary PLI table file named `pli_learn.tab`.

We call this file 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 what ever ACC capabilities are specified in a first PLI table file, or other means of specifying ACC capabilities, so that you enable only the ACC capabilities you need in subsequent simulations.

You should look at the contents of the `pli_learn.tab` file that VCS writes to see what ACC 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 ACC 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 ACC capabilities on a signal by signal basis.

The following line in this file specifies that during simulation the ACC 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 ACC capabilities were used during the simulation.

```
//!VCS_LEARNED: NO_DYNAMIC_ACCESS_PERFORMED
```

This indicates that only static information was accessed through ACC capabilities and there was no value change information during simulation.

These comments mean there is a potentially significant performance gain when you apply the ACC capabilities in the `pli_learn.tab` file.

Compiling to Enable Only the ACC Capabilities You Need

After you have run the simulation to learn what ACC 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 ACC capabilities that you need.

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 ACC capabilities you need, you will need to have VCS write another one, by including the `+vcs+learn+pli` runtime option and then compile your design again with the `+applylearn` option.

Limitations

VCS is not able to keep track of all ACC capabilities. The capabilities it can keep track of, 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 any of the following compile-time options:

- `+cli` because this option does not specify what information will be accessed through CLI commands.

- `+multisource_int_delays` or `+transport_int_delays` because interconnect delays need global ACC capabilities.

If you enter the `+applylearn` compile-time option more than once on the `vcs` command line, VCS ignores all but the first one.

Using VPI Routines

To enable VPI capabilities in VCS, you must include the `+vpi` compile-time option. For example:

```
vcs +vpi test.v -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 17-31](#).

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 17-32](#).

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 can't 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:

<code>cbEndOfSimulation</code>	<code>cbError</code>	<code>cbPliError</code>
<code>cbTchkViolation</code>	<code>cbSignal</code>	<code>cbForce</code>

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

Support for the `vpi_register_systf` Routine

VCS supports the `vpi_register_systf` VPI access routine. To use it 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, ← entry here
    0 /* Last Entry */
};
```

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` compile-time option. For example:

```
vcs any.c any.v -use_vpiobj vpi_user.c +cli+3 +vpi
```

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, which call PLI ACC routines. 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 ACC routines. The syntax has not changed; you use the same syntax for enabling ACC and VPI routines.

This PLI table file is for an example file named `set_mipd_delays_vpi.c` that is available with *The Verilog PLI Handbook* by Stuart Sutherland, Kluwer Academic Publishers, Boston, Dordrecht, and London.

Integrating a VPI Application With VCS

If you create one or more shared libraries for a VPI application, the application should not contain the array named `vlog_startup_routines`.

Instead you 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 pathname 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.

Writing Your Own `main()` Routine

You write your own `main()` routine if you wrote your PLI application in C++ code or if your standard C code application does some processing before starting the `simv` executable.

When you write your own `main()` routine you must include the `-e` compile-time option on the `vcs` command line. The syntax is as follows:

```
-e new_name_for_main
```

For example:

```
vcs -P my_main.tab my_main.cc -e SimvMain source.v
```

The contents of `my_main.cc` is as follows:

```
#if defined(__cplusplus)
extern "C" {
#endif

extern int SimvMain(int argc, char *argv[]);
extern void vcs_atexit (void(*pfun)(int code));

#if defined(__cplusplus)
}
#endif
```

```

static void do_cleanup(int code)
{
    /* do simv post-processing work */
}

int main(int argc, char *argv[])
{
    /* Startup code (if any) goes here. */

    vcs_atexit(do_cleanup); /* Register callback */

    SimvMain(argc, argv); /* Run simv */

    return 0; /* Never gets here */
}

```

Note that SimvMain does not return, it calls exit() directly. If you need to do any post-processing, you can register at-exit functions using vcs_atexit().

The function you pass to vcs_atexit() is called with the exit status. If you make multiple calls to vcs_atexit(), your functions are called in the reverse order of registration.

Note:

You cannot use this feature when using the VCS/ SystemC cosimulation interface.

Using the PLI

17-36

18

DirectC Interface

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 kinds 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, then 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 like you call (or enable) a Verilog function or Verilog task.

This chapter describes the DirectC interface in the following sections:

- [Using Direct C/C++ Function Calls](#)
- [Using Direct Access](#)
- [Using Abstract Access](#)
- [Enabling C/C++ Functions](#)
- [Environment Variables](#)
- [Extended BNF for External Function Declarations](#)

Using Direct C/C++ Function Calls

To enable a direct call of a C/C++ function during simulation, do the following:

1. Declare the function in your Verilog code.
2. Call the function in your Verilog code.
3. Compile your Verilog 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. This is because, in the Verilog environment, they become analogous to Verilog tasks as well as functions. Verilog tasks are like void C functions in that they don't return a value. Verilog tasks do however have input, output, and inout arguments, whereas C function parameters do not have explicitly declared directions. See ["Declaring the C/C++ Function" on page 18-6](#).

There are two access modes for C/C++ function calls. These modes don't 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 ["Using Direct Access" on page 18-20](#)
- 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 ["Using Abstract Access" on page 18-29](#)

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 the 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 18-1 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`. These keywords specify in a C/C++ function the same thing that they specify in a Verilog task; see Table 18-2.

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 18-1 C/C++ Function Return Types

Return Type	What it 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 18-2 C/C++ Function Argument Directions

keyword	What it specifies
<code>input</code>	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 keyword <code>input</code> .
<code>output</code>	The C/C++ function can only write the value or address of the argument.
<code>inout</code>	The C/C++ function can both read and write the value or address of the argument.

Table 18-3 C/C++ Function Argument Types

keyword	What it specifies
<code>real</code>	The C/C++ function reads or writes the address of a Verilog real data type.
<code>reg</code>	The C/C++ function reads or writes the value or address of a Verilog reg.
<code>bit</code>	The C/C++ function reads or writes the value or address of a Verilog reg in two state simulation.
<code>int</code>	The C/C++ function reads or writes the address of a C/C++ int data type.
<code>pointer</code>	The C/C++ function reads or writes the address that a pointer is pointing to.
<code>string</code>	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 `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 18-20](#). 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 passed 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 new 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 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:

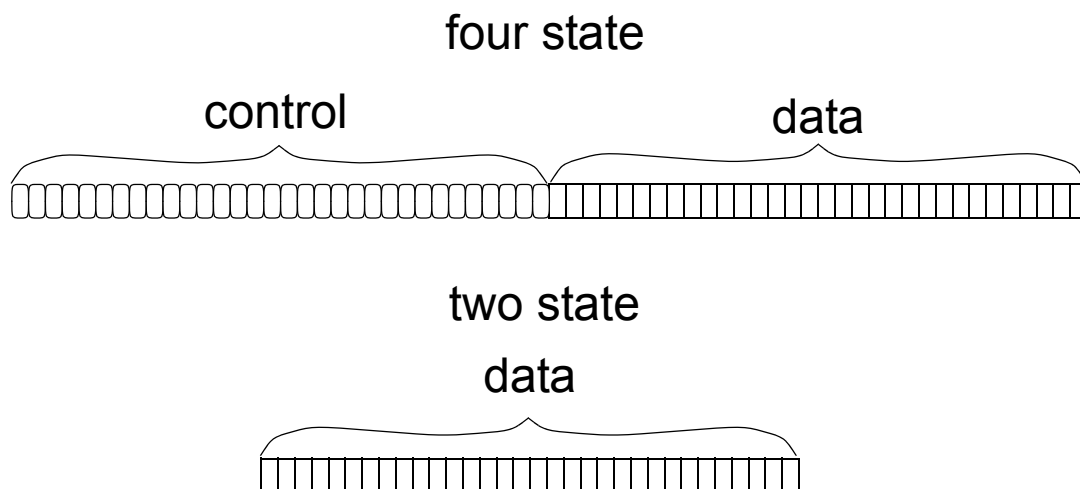
```
typedef unsigned int U;  
typedef struct { U c; U d;} vec32;
```

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.

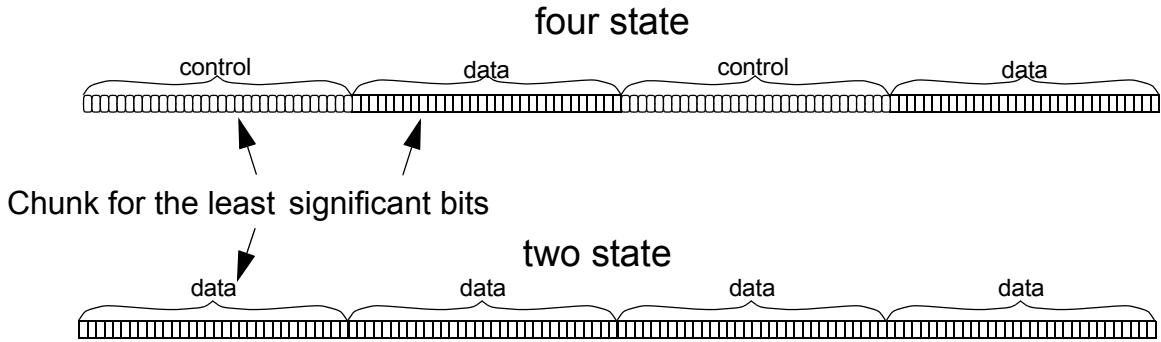
Figure 18-4 Storing Vector Values



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.

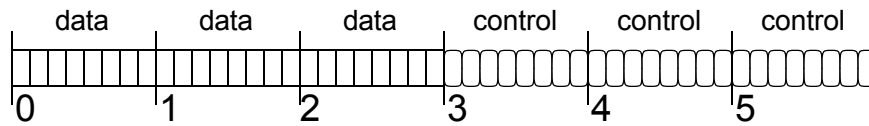
For long vectors the chunk for the least significant bits come first, followed by the chunks for the more significant bits.

Figure 18-5 Storing Vector Values of More Than 32 Bits



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 18-6 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, 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_ConvertTo String(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 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:

<code>int</code>	<code>int*</code>	<code>double*</code>	<code>void*</code>	<code>void**</code>
<code>char*</code>	<code>char**</code>	<code>scalar</code>	<code>scalar*</code>	
<code>U*</code>	<code>vec32</code>	<code>UB*</code>		

Some of these type identifiers are standard C/C++ types; the ones that aren't 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 18-7 For Input Arguments

argument type	C/C++ formal parameter data type	Passed by
<code>int</code>	<code>int</code>	value
<code>real</code>	<code>double*</code>	reference
<code>pointer</code>	<code>void*</code>	value
<code>string</code>	<code>char*</code>	value
<code>bit</code>	<code>scalar</code>	value
<code>reg</code>	<code>scalar</code>	value
<code>bit []</code> - 1-32 bit wide vector	<code>U</code>	value
<code>bit []</code> - open vector, any vector wider than 32 bits	<code>U*</code>	reference
<code>reg []</code> - 1-32 bit wide vector	<code>vec32*</code>	reference
<code>array []</code> - open vector, any vector wider than 32 bits	<code>UB*</code>	reference

Table 18-8 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
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);
```

Here 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);
```

Here 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 );
```

Here 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. Here 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);
```

Here 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);
```

Here the function named `incr`, that does not return a value, has an argument declared as `inout`. No bit-width is specified for it 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);
```

Here 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)
```

Here the parameters `in` and `out` are pointers to type `U`. Pointers because their corresponding arguments are larger than 32 bits and type `U` because the 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);
```

Here 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)
```

Here 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);
```

Here 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)
```

Here 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 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 don't 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 indx;
    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 don’t 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;
```

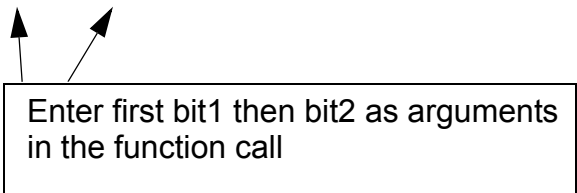
Using vc_handle

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 bear in mind that they can be overwritten in Verilog. So 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" on page 18-77](#).

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
```


Here 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("\n i1=%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("mempout : %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 use 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 use 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
```

Here 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 C/C++ function `passback` 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" string FullPath(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
    s2v(FullPath("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: %dbits, 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:

```

extern void big_2state( input bit [31:0] r1,
                      input bit [32:0] r2);

module test;
reg [31:0] r1;
reg [32:0] r2;
initial
begin
r1=4294967295;
r2=33'b1000000000000000000000000000000010;
big_2state(r1,r2);
end
endmodule

```

Here the Verilog code declares a 32-bit bit vector, `r1`, and a 33-bit bit vector, `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 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 the 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

```


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
```

Here 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;
    }
}

```

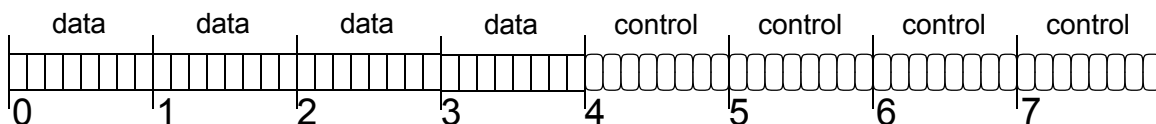
The purpose of the C/C++ function `mem_doer` is to copy the four elements in Verilog memory `memory1` into the 32 elements of `memory2`.

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.

Why copy a total of 64 bytes? 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.

Figure18-9 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 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'bz00000000xxxxxxxx11111111;
$display("memory1 [0] = %0b\n", memory1[0]);
mem_add_doer(memory1);
$display("\nmemory1 [3] = %0b", memory1[3]);
end
endmodule
```

Here 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.

Here 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_doer(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_doer` 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] = z00000000xxxxxxxx11111111
```

```
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] = z00000000xxxxxxxx11111111
```

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
```

Here in this 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);
}
```

Here 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
```

Here 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 here 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 is 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 output 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;
```


So 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 we, as shown here, specify that it have only one element. The `vc_get4stMemoryVector` routine copies the data from the Verilog memory element (here 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 don't 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 indx, 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 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 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 is for passing values to from memory element 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 of 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 how to use this routine to fill a reg and a memory these 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 is to translate the multiple indices into a single index of a linearized array, This routine does this. 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 suum of all elements from a 2-dimensional slice
of a 4-dimensional array */
int getSlice(vc_handle vh_array, vc_handle vh_indx1,
vc_handle vh_indx2) {
```

```

int sum = 0;
int i1, i2, i3, i4, indx;

i1 = vc_getInteger(vh_indx1);
i2 = vc_getInteger(vh_indx2);
/* 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++) {

        indx = vc_Index(vh_array, i1, i2, i3, i4);
        sum += vc_getMemoryInteger(vh_array, indx);
    }
}
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 18-1 summarizes all the access routines described in the previous section

Table 18-1 Summary of Access Routines

Access Routine	Description
int vc_isScalar(vc_handle)	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.
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.
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.
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.
int vc_is2state(vc_handle)	This routine does the opposite of the vc_is4state routine.
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 to a memory.
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.
int vc_width(vc_handle)	Returns the width of a vc_handle.

Access Routine

```
int
vc_arraySize(vc_handle)

scalar
vc_getScalar(vc_handle)

void
vc_putScalar(vc_handle,
scalar)

char
vc_toChar(vc_handle)

int
vc_toInteger(vc_handle)

char
*vc_toString(vc_handle)

char
*vc_toStringF(vc_handle,
char)

void
vc_putReal(vc_handle,
double)

double
vc_getReal(vc_handle)

void
vc_putValue(vc_handle,
char *)

void
vc_putValueF(vc_handle,
char, char *)

void
vc_putPointer(vc_handle,
void*)
void
*vc_getPointer(vc_handle)
```

Description

Returns the number of elements in a memory.

Returns the value of a scalar reg or bit.

Passes the value of a scalar reg or bit to a `vc_handle` by reference.

Returns the 0, 1, x, or z character.

Returns an int value for a `vc_handle` to a scalar bit or a vector bit of 32 bits or less.

Returns a string that contains the 1, 0, x, and z characters.

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'.

Passes by reference a real (double) value to a `vc_handle`.

Returns a real (double) value from a `vc_handle`.

This function passes, by reference through the `vc_handle`, a value represented as a string containing the 0, 1, x, and z characters.

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'.

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.

Access Routine

```
void  
vc_StringToVector(char *,  
vc_handle)
```

```
void  
vc_VectorToString(vc_handl  
e, char *)
```

```
int  
vc_getInteger(vc_handle)
```

```
void  
vc_putInteger(vc_handle,  
int)
```

```
vec32  
*vc_4stVectorRef(vc_handl  
e)
```

```
U  
*vc_2stVectorRef(vc_handl  
e)
```

```
void  
vc_get4stVector(vc_handle  
, vec32 *)  
void  
vc_put4stVector(vc_handle  
, vec32 *)
```

```
void  
vc_get2stVector(vc_handle  
, U *)  
void  
vc_put2stVector(vc_handle  
, U *)
```

```
UB  
*vc_MemoryRef(vc_handle)
```

```
UB  
*vc_MemoryElemRef(vc_hand  
le, U indx)
```

Description

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).

Converts a vector value to a string value.

Same as `vc_toInteger`.

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.

Returns a `vec32` pointer to a four state vector. Returns NULL if the specified `vc_handle` is not to a four-state vector reg.

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.

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.

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.

Returns a pointer of type `UB` that points to a memory in Verilog.

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.

Access Routine

```
scalar  
vc_getMemoryScalar(vc_handle, U indx)
```

```
void  
vc_putMemoryScalar(vc_handle, U indx, scalar)
```

```
int  
vc_getMemoryInteger(vc_handle, U indx)
```

```
void  
vc_putMemoryInteger(vc_handle, U indx, int)
```

```
void  
vc_get4stMemoryVector(vc_handle, U indx, vec32 *)
```

```
void  
vc_put4stMemoryVector(vc_handle, U indx,  
vec32 *)
```

```
void  
vc_get2stMemoryVector(vc_handle, U indx, U *)
```

```
void  
vc_put2stMemoryVector(vc_handle, U indx, U *)
```

```
void  
vc_putMemoryValue(vc_handle, U indx, char *)
```

```
void  
vc_putMemoryValueF(vc_handle, U indx, char, char *)
```

```
char  
*vc_MemoryString(vc_handle, U indx)
```

Description

Returns the value of a one bit memory element.

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.

Returns the integer equivalent of the data bits in a memory element whose bit-width is 32 bits or less.

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.

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.

Copies Verilog data from a vec32 array to a Verilog memory element.

Copies the data bytes, but not the control bytes, from a Verilog memory element to an array in your C/C++ function.

Copies Verilog data from a U array to a Verilog memory element. This routine is used in the previous example.

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.

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.

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.

Access Routine

```
char  
*vc_MemoryStringF(vc_handle, U indx, char)
```

```
void  
vc_FillWithScalar(vc_handle, scalar)
```

```
char  
*vc_argInfo(vc_handle)
```

```
int vc_Index(vc_handle, U,  
...)
```

```
int vc_Index2(vc_handle,  
U, U)
```

```
int vc_Index3(vc_handle,  
U, U, U)
```

```
U vc_mdaSize(vc_handle, U)
```

Description

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'.

This routine fills all the bits of a reg, bit, or memory with all 1, 0, x, or z values (you can choose only one of these four values).

Returns a string containing the information about the parameter in the function call in your Verilog source 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.

Specialized version of `vc_Index()` where the two U parameters are the indices in a two-dimensional array.

Specialized version of `vc_Index()` where the two U parameters are the indexes in a three-dimensional array.

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_hdrs.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
<code>extern</code> (no mode specified)	direct access	abstract access
<code>extern "A"</code>	abstract access	abstract access
<code>extern "C"</code>	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/vcs -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 `$VCS_HOME/include/DirectC.h` file to you current directory. VCS will always look there for it.
- Establish a link in the current directory to the `$VCS_HOME/include/DirectC.h` file.

- Include the `-CC` option as follows:

```
-CC "-I$VCS_HOME/include"
```

Useful Compile-Time Options

VCS has other compile-time options that are not specially for DirectC but you might find them useful when enabling and calling C/C++ functions in your Verilog source code.

`-cc`

Specifies the C compiler (default is `cc` in your search path)

`-cpp`

Specifies the C++ compiler (default is `CC` in your search path)

`-ld`

Specifies the linker for final linking to build `simv` (default is the same as the C++ compiler)

Note:

Don't specify incompatible C++ compiler and linker (by specifying `-cpp` and `-ld` simultaneously). That may result in compile failure with unrecognized symbols.

These options may be necessary on the `vcs` command line. If you include object files on the command line, you must include the options to specify which compiler and linker generated these object files.

Environment Variables

The following environment variables can be useful with DirectC. Bear in mind that command line options override environment variables.

VCS_CC

Specifies the C compiler for VCS, same as the `-cc` compile-time option.

VCS_CPP

Specifies the C++ compiler for the PLI and cmodule code compilation, same as the `-cpp` compile-time option.

VCS_LD

Specifies the linker.

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 command line options `+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; 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.

19

Using the VCS / SystemC Cosimulation Interface

The VCS / SystemC Cosimulation 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 . By default, when you use the interface, VCS runs its own SystemC simulator. No setup is necessary.

.You also have the option of installing the OSCI SystemC simulator and having VCS run it to cosimulate using the interface. See [“Using a Customized SystemC Installation”](#) on page 19-43.

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 Cosimulation 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 testbenches 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 Cosimulation Interface creates the necessary infrastructure to cosimulate 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 cosimulation, the VCS / SystemC Cosimulation Interface is responsible for:

- Synchronizing the SystemC kernel and VCS
- Exchanging data between the two environments

Notes:

- There are examples of Verilog instantiated in SystemC and SystemC instantiated in Verilog in the `$VCS_HOME/doc/examples/osci_dki` directory.
- The interface supports the following compilers:
 - Linux: gnu 3.3.6 compiler
 - Solaris: SC 8.0, and gcc 3.3.6 (default) and 3.4.6
- The VCS / SystemC Cosimulation Interface supports 32-bit simulation as well as 64-bit (VCs flag `-full64`) simulation. Do not use the `-comp64` compile-time options with the interface.
- The gcc 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.

The usage models for the VCS / SystemC Cosimulation Interface, depending on the type of cosimulation you want to perform. This chapter describes these models in the following sections:

- [Usage Scenario Overview](#)
- [Verilog Design Containing SystemC Leaf Modules](#)
- [SystemC Designs Containing Verilog Modules](#)
- [Using a Port Mapping File](#)
- [Using a Data Type Mapping File](#)
- [Debugging the SystemC Portion of a Design](#)
- [Transaction Level Interface](#)

- [Using a Customized SystemC Installation](#)

Usage Scenario Overview

The usage models for the VCS /SystemC Cosimulation Interface are:

- Verilog designs containing SystemC modules
- SystemC designs containing Verilog modules

The major steps involved to create a simulation for each of these design scenarios are:

1. Analyze the SystemC and Verilog modules from the bottom of the design to the top.
2. For Verilog designs containing SystemC modules:
 - Use the `syscan file.cpp:model` command to analyze SystemC modules used in the Verilog domain.
 - Use the `syscan f.cpp...` command to compile other SystemC modules in the design.
 - Use the `vlogan` command to analyze Verilog files.
 - Use the `vcs -sysc` command to build the simulation.
3. For SystemC designs containing Verilog modules:
 - Use the `vlogan -sc_model` command to analyze Verilog files containing modules used in the SystemC domain.
 - Use the `syscan f.cpp...` command to compile SystemC files.
 - Use the `syscsim` command to build the simulation.

Note:

There are examples of Verilog instantiated in SystemC, and SystemC instantiated in Verilog, in the `$VCS_HOME/doc/examples/osci_dki` directory.

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_biguint` 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.

Inout ports that cross the cosimulation boundary between SystemC and Verilog must observe the following restrictions:

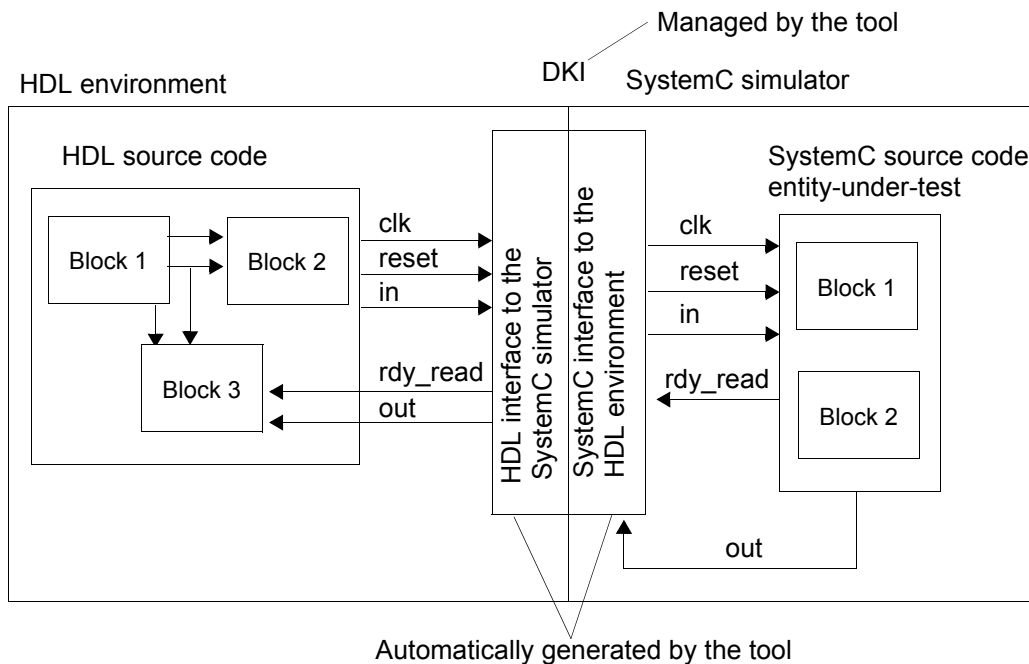
- SystemC port types must be `sc_inout_rv<>` or `sc_inout_resolved` and must be connected to signals of type `sc_signal_rv<>` or `sc_signal_resolved`.
- Verilog port types must be `bit_vector` or `bit`.
- You need to create a port mapping file, as described in [“Using a Port Mapping File” on page 19-26](#), to specify the SystemC port data types as `sc_lv` (for a vector port) or `sc_logic` (for a scalar port).

Verilog Design Containing SystemC Leaf Modules

To cosimulate a Verilog design that contains SystemC and Verilog modules, you need to import one or more SystemC instances into the Verilog design. Using the VCS / SystemC Cosimulation Interface, you generate a wrapper and include it in the Verilog design for each SystemC instance. The ports of the created Verilog wrapper are connected to signals that are attached to the ports of the corresponding SystemC modules.

Figure 19-1 illustrates VCS DKI communication.

Figure 19-1 VCS DKI Communication of an Verilog Design Containing SystemC Modules



Input Files Required

To run a cosimulation 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:
 - A Verilog top-level simulation that instantiates the interface wrapper and other Verilog modules. These wrapper files are generated by a utility and you don't need to do anything to these files (see [“Generating the Wrapper for SystemC Modules” on page 19-8](#) and [“Instantiating the Wrapper and Coding Style” on page 19-11](#)).
 - 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 19-26](#).
- 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 19-27](#).

Generating the Wrapper for SystemC Modules

You use the `syscan` utility to generate the wrapper and interface files for cosimulation. This utility creates the `csrc` directory in the current directory, just like VCS does when you include compile-time options for incremental compilation. The `syscan` utility writes the wrapper and interface files in subdirectories in the `./csrc` directory.

There is nothing you need to do to the files that `syscan` writes. VCS knows to look for them when you include the compile-time option for using the interface. See [“Compiling a Verilog Design Containing SystemC Modules” on page 19-14](#).

The syntax for the `syscan` command line is as follows:

```
syscan [options] filename[:modulename]  
[filename[:modulename]]*
```

Here:

```
filename[:modulename] [filename[:modulename]]*
```

This is how you specify all the SystemC files in the design. There is no limit to the number of files. The entries for the SystemC files that contain modules, which you want to instantiate, also include a colon `:` followed by the name of the module. 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 omit `-cpp path_to_the_compiler`, the environment finds 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++
```

`-port port_mapping_file`

Specifies a port mapping file. See [“Using a Port Mapping File” on page 19-26](#).

`-Mdir=directory_path`

Works the same way that the `-Mdir` VCS compile-time option. If you are using the `-Mdir` option with VCS, you should use the `-Mdir` option with `syscan` to redirect the `syscan` output to the same location that VCS uses.

`-help|-h`

Displays the command line syntax, options, and example command lines.

`-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 are encountering errors or are interested in the flow that builds the design.

-vcsi

Prepares all SystemC interface models for simulation with VCSi.

-f *filename*

Specifies a file containing one or more *filename[:modulename]* entries, as if these entries were on the command line.

-verilog | -vhdl

Indicates which domain the interface models should be prepared for. `-verilog` is the default.

Note:

You don't specify the data type mapping file on the command line, See ["Using a Data Type Mapping File" on page 19-27](#).

The following example generates a Verilog wrapper:

```
syscan -cflags "-g" sc_add.cpp:sc_add
```

Instantiating the Wrapper and Coding Style

You instantiate the SystemC wrapper just like a Verilog module. For example, consider the following SystemC module in a file named `stimulus.h`:

```
SC_MODULE(stimulus) {
    sc_out<sc_logic>    reset;
    sc_out<sc_logic>    input_valid;
    sc_out<sc_lv<32> > sample;
    sc_in_clk           clk;

    sc_int<8>           send_value1;
    unsigned            cycle;

    SC_CTOR(stimulus)
        : reset("reset")
        , input_valid("input_valid")
        , sample("sample")
        , clk("clk")

    {
        SC_METHOD(entry);
        sensitive_pos(clk);
        send_value1 = 0;
        cycle       = 0;
    }
    void entry();
};
```

The Verilog model is display:

```
File: display.v
module display (output_data_ready, result);
    input        output_data_ready;
    input [31:0] result;

    integer counter;
```

```

initial
begin
    counter = 0;
end

always @(output_data_ready)
begin
    counter = counter + 1;
    $display("Display : %d", result);
    if (counter >= 24)
    begin
        $finish;
    end
end
endmodule

```

You instantiate the SystemC model as follows in the Verilog part of the design:

File: tb.v

```

module testbench ();

    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));

```

```
...
end module
```

Controlling Time Scale and Resolution in a SystemC Module Contained in a Verilog Design

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);
            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();
```

Compiling a Verilog Design Containing SystemC Modules

To compile your Verilog design, include the `-sysc` compile-time option. For example:

```
vcs -sysc tb.v display.v
```

When you compile with this option, VCS looks in the `csrc` directory for the subdirectories containing the interface and wrapper files needed to instantiate the SystemC design in the Verilog design.

Using GNU Compilers on Sun Solaris

On Solaris the default compiler is Sun Forte CC. You can specify a different compiler with `-cpp` and `-cc` compile-time options. The interface supports the gcc 3.3.6 (default) and 3.4.6 compilers.

If you use the `-cpp g++` option on the `syscan` command line, you must also use it on every command line that compiles C++ source. For example:

```
syscan -cpp g++ sc_add.cpp:sc_add
```

```
syscan -cpp g++ sc_sub.cpp multiply.cpp display.cpp
```

```
vcs -cpp g++ -sysc top.v dev.v
```

If you use a full path to a C++ compiler, you have to supply the path to the `cc` compiler on the VCS command line as well:

```
syscan -cpp /usr/bin/g++ sc_add.cpp:sc_add
```

```
syscan -cpp /usr/bin/g++ sc_sub.cpp multiply.cpp display.cpp
```

```
vcs -cc /usr/bin/gcc -cpp /usr/bin/g++ -sysc top.v dev.v
```

Using GNU Compilers on Linux

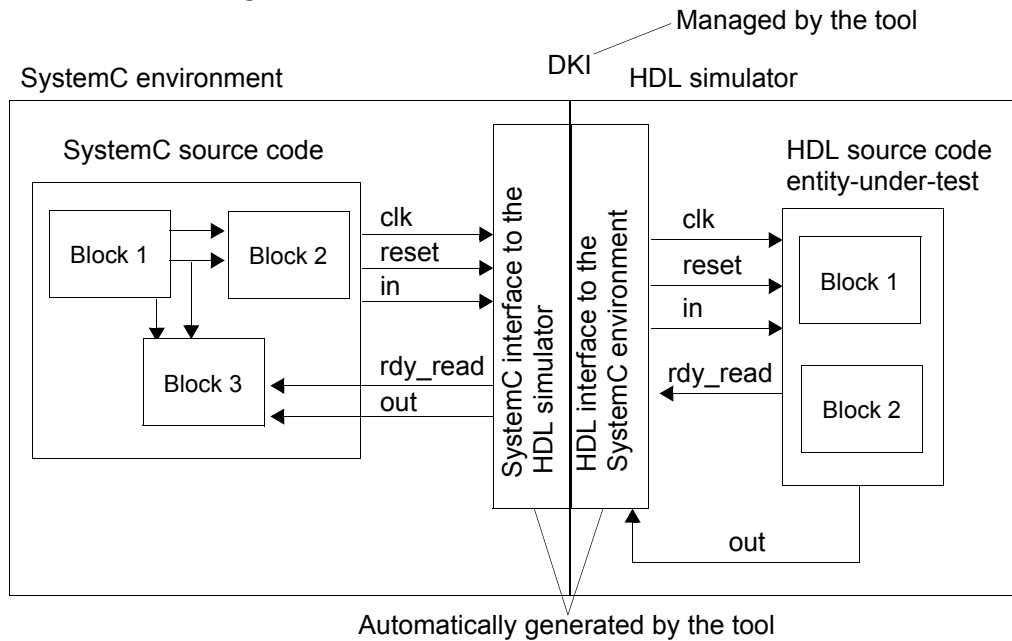
On Linux the default compiler is gcc. You can specify a different compiler with the `-cpp` and `-cc` compile-time options. The interface supports the gcc 3.3.6 compiler.

SystemC Designs Containing Verilog Modules

To cosimulate a SystemC design that contains Verilog modules, you import one or more Verilog instances into the SystemC design. Using the VCS / SystemC Cosimulation Interface, you generate a wrapper and include it in the SystemC design for each Verilog instance. The ports of the created SystemC wrapper are connected to signals that are attached to the ports of the corresponding Verilog modules.

Figure 19-2 illustrates the VCS direct kernel interface (DKI) communication.

Figure 19-2 VCS DKI Communication of SystemC Design Containing Verilog Modules



Input Files Required

To run cosimulation with a SystemC design containing Verilog modules, you need to provide the following files:

- Verilog source code (.v extensions)
 - You can directly write the entity-under-test Verilog code or generate it with other tools. The Verilog description represented by the entity-under-test can be Verilog code of any complexity (including hierarchy) and can use any language feature VCS supports.
 - Any other 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 19-26](#).
- 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 19-27](#).

Generating the Wrapper

You use the `vlogan` utility with the `-sc_model` option to generate and build the wrapper and interface files for Verilog modules for cosimulation. This utility creates the `./csrc` directory in the current directory, just like VCS does when you include compile-time options for incremental compilation. The `vlogan` utility writes the wrapper and interface files in subdirectories in the `./csrc` directory.

There is nothing you need to do to the files that `vlogan` writes. VCS knows to look for them when you include the compile-time option for using the interface. See [“Compiling a Verilog Design Containing SystemC Modules” on page 19-14](#).

The syntax for the `vlogan` command line is as follows:

```
vlogan -sc_model modulename file.v  
[-cpp path_to_the_compiler]  
[-sc_portmap port_mapping_file]  
[-Mdir=directory_path] [-V]
```

Here:

```
-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. See [“Using a Port Mapping File” on page 19-26](#).

`-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.

To generate the wrapper and interface files for a Verilog module named `adder`, in a Verilog source file named `adder.v`, instantiated in SystemC code in `top.cpp`, you would enter the following:

```
vlogan -sc_model adder -sc_portmap the.map adder.v
```

Instantiating the Wrapper

You instantiate a Verilog module in your SystemC code like a SystemC module. For example, consider the following Verilog module in a file called `adder.v`:

```
module adder (value1, value2, result);
    input [31:0] value1;
    input [31:0] value2;
    output [31:0] result;
    reg [31:0] result_reg;

    always @(value1 or value2)
    begin
        result_reg <= value1 + value2;
    end

    assign result = result_reg;

endmodule
```

The module name is `adder`. You instantiate it in your SystemC code in `main.cpp` as follows:

```
#include <systemc.h>
#include "adder.h"
int sc_main(int argc, char *argv[]){
    sc_clock clock ("CLK", 20, .5, 0.0);
    sc_signal<sc_lv<32> > value1;
    sc_signal<sc_lv<32> > value2;
    sc_signal<sc_lv<32> > result;

    // Verilog adder module
    adder adder1("adder1");
    adder1.value1(value1);
    adder1.value2(value2);
    adder1.result(result);

    sc_start(clock, -1);
}
```

One of the generated files is `modulename.h`, which you should include in your `top.cpp` file.

Compiling a SystemC Design Containing Verilog Modules

When you compile your design, you must include the hierarchy path to the SystemC wrapper instances on your design compilation command line. For example:

```
syscsim dev.v other_C++_source_files compile-time_options
adder=adder1
```

In this example, dev.v might contain Verilog code utilized by the adder.v module above.

When you compile with this option, VCS looks in the ./csrc directory for the subdirectories containing the interface and wrapper files needed to connect the Verilog and SystemC parts of the design.

Elaborating the Design

When SystemC is at the top of the design hierarchy and you instantiate Verilog code in the SystemC code, the elaboration of the simulation is done in the following two steps:

1. The first step is to create a temporary simulation executable that contains all SystemC parts but does not yet contain any Verilog parts. VCS then starts this temporary executable to find out which Verilog instances are really needed. All SystemC constructors and end_of_elaboration() methods are executed; however, simulation does not start.
2. VCS creates the final version of the simv file containing SystemC as well as all HDL parts. The design is now fully elaborated 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:

```
INFO: Exiting prematurely since $SYSTEMC_ELAB_ONLY is set
```

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 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:

```
module constructor:
  if (! hdl_elaboration_only()) {
    ... open log file for coverage data ...
  }
  module destructor:
  if (! hdl_elaboration_only()) {
    ... close log file for coverage data ...
  }
}
```

Considerations for Export DPI Tasks

When you want to call export "DPI" tasks from the SystemC side of the design you need to do either one of the following two steps.

Use `syscan -export_DPI <function-name>`

Register the name of all export DPI functions and tasks prior to the final `syscsim` call. 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, 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:

```
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 elaborate the simulation

```
syscan -export_DPI task1
syscan -export_DPI function3
...
syscsim ...
```

Note that task2 is not specified because it is an import "DPI" task.

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`).

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 simulator kernel. All arguments following `-systemcrun` go only to `sc_main()`. All arguments following `-verilogrun` or `-vhdlrun` 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 -ucli g -verilogrun c d -systemcrun e f
```

Function `sc_main()` receives arguments "a b e f g". The VCS simulator kernel receives arguments "c d -ucli."

Using GNU Compilers on SUN Solaris

On Solaris the default compiler is Sun Forte CC. You can specify a different compiler with `-cpp` and `-cc` compile-time options. The interface supports the gcc 3.3.6 compiler.

If you use the `-cpp g++` option on the interface analysis command line, you must also use it on every command line that compiles C++ source:

```
vlogan -cpp g++ display.v -sc_model display
```

```
syscsim -cpp g++ -sysc main.cpp a.cpp display=display1
```

If you use a full path to a C++ compiler, you have to supply the path to the cc compiler on the command line that builds the simulation as well:

```
vlogan -cpp /usr/bin/g++ display.v -sc_model display
```

```
syscsim -cc /usr/bin/gcc -cpp /usr/bin/g++ -sysc main.cpp  
a.cpp
```

Using GNU Compilers on Linux

On Linux the default compiler is gcc. You can specify a different compiler with the `-cpp` and `-cc` compile-time options. The interface supports the gcc 3.3.6 compiler.

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 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 x and y. 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 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 (`bitvector` 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 19-1 Port Mapping File

#	Port name	Bits	Verilog type	SystemC type
	in1	8	signed	sc_int
	in2	8	bit_vector	sc_lv
	clock	1	bit	sc_clock
	out1	8	bit_vector	sc_uint
	out2	8	bit_vector	sc_uint

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 overrides 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 19-2 shows an example of a data type mapping file.

Example 19-2 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

Debugging the SystemC Portion of a Design

To debug just the SystemC code in the mixed simulation, do the following:

1. Run `syscan` with the `-cflags "-g"` option to build the SystemC source code for debugging.
2. Start the C++ debugger on the `simv` executable file as follows:

- If you are using the Sun Forte compiler:

```
dbx ./simv
```

- If you are using the Gnu compiler on Solaris or Linux:

Run both `syscan` and `VCS` with the `-cpp path` option.

```
gdb ./simv
```

You can now set and stop at breakpoints in your SystemC code.

Debugging the Verilog Code

To debug the Verilog code, create the `simv` executable with the `-RI` option to start VirSim for interactive debugging, for example if you instantiate SystemC code in Verilog code:

```
vcs -sysc -R -o simv top.v
```

If you instantiate Verilog code in SystemC:

```
syscsim -R -o simv dev.v
```

Debugging Both the Verilog and SystemC Portions of a Design

To debug both the SystemC and Verilog portions of your design:

1. Run `syscan` with the `-cflags "-g"` option to build the SystemC source code for debugging.
2. Include the `-debug_all` compile-time options on the `vcs` or `syscsim` command line to compile the Verilog part of the design for post-processing debug tools and for Verilog source code debugging.

To compile and interactively debug a Verilog design containing SystemC modules, enter command lines like the following:

```
vcs -sysc -debug_all top.v
```

To compile and interactively debug a SystemC design containing Verilog modules, enter command lines like the following:

```
syscsim -I -line
```

3. Start the C++ debugger on the `simv` executable file. As DVE is already running the `simv` executable, you must attach your debugger to the `simv` process.
4. To find the `simv` executable process ID, execute the following command:

```
% ps -e | grep simv
12216 pts/1      0:00 simv
```

5. You then can launch your debugger as outlined above, but provide the process ID from the ps command as the third argument to the debugger:

```
% gdb ./simv 12216
```

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 the user 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 and have 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 and 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 use 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 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 19-40](#) 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.

There is one special case here. If the methods of the SystemC interface class use reference parameters, for example `my_method(int& par)`, then you need to mark this parameter as `inout&` parameter 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
```

The 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 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. If, on the other hand, 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, See [“Instantiation and Binding” on page 19-38](#).

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	char
inout output	byte	char*
input	shortint	short int
inout output	shortint	short int*
input	int	int
inout output	int	int*
input	longint	long long

SystemVerilog			SystemC
inout output	longint		long long*
input	real		double
inout output	real		double*
input	shortreal		float
inout output	shortreal		float*
input	chandle		void*
inout output	chandle		void**
input	string		char*
inout output	string		char**
input	bit		unsigned char
inout output	bit		unsigned char*
input	logic		unsigned char
inout output	logic		unsigned char*

For the integral data types in the above table, the signed and unsigned qualifiers are allowed and map 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

Using the Built-in SystemC Simulator

VCS contains a built-in SystemC 2.0.1 which it uses by default. No setup is necessary to use this simulator (in earlier releases, the interface required the `SYSTEMC` environment variable).

Supported Compilers

The following compilers are supported:

- Linux: gcc 3.3.4 (default) and 3.4.6
- Solaris: SC 6.2, gcc 3.3.4 (default) and 3.4.6

Compiling Source Files

If you need to compile source files, which include `systemc.h`, in your own environment and not with the `syscan` script, then add compiler flag `-I$VCS_HOME/include/systemc`.

Using a Customized SystemC Installation

You can install the OSCI SystemC simulator and instruct VCS to use it for Verilog/SystemC cosimulation. To do so you need to:

- Obtain OSCI SystemC version from www.systemc.org
- Set the `SYSTEMC` environment variable to the OSCI SystemC installation path, as shown below:

```
% setenv SYSTEMC /net/user/download/systemc-2.0.1
```

- Replace following SystemC files

```
$SYSTEMC/src/systemc/kernel/sc_simcontext.cpp  
$SYSTEMC/src/systemc/kernel/sc_simcontext.h
```

with following VCS files:

```
$VCS_HOME/etc/systemc-2.0.1/sc_simcontext.cpp and  
$VCS_HOME/etc/systemc-2.0.1/sc_simcontext.h
```

- Follow the installation instructions provided by OSCI (see `$SYSTEMC/INSTALL` file) and build a SystemC library. Use `../configure i686-pc-linux-gnu` call for 32-bit linux installation, and `../configure` call for 32-bit solaris installation.

- Set the `SYSTEMC_OVERRIDE` environment variable to the user defined OSCI SystemC library installation path, as shown below:

```
% setenv SYSTEMC_OVERRIDE /net/user/systemc-2.0.1
```

- Header files must exist in the `$SYSTEMC_OVERRIDE/include` directory and the library file `libsystemc.a` in `$SYSTEMC_OVERRIDE/lib-linux/`, and the `$SYSTEMC_OVERRIDE/lib-gccsparcos5/` directories.

Note:

- VcsSystemC 2.0.1 is binary compatible with OSCI SystemC 2.0.1.
- VcsSystemC 2.1 is binary compatible with OSCI SystemC 2.1 Beta as of October 2004, but not compatible with OSCI SystemC 2.1.v1.
- VcsSystemC 2.2 is binary compatible with OSCI SystemC 2.2 Beta as of 5th June 2006.

Compatibility with OSCI SystemC 2.0.1 and SystemC 2.1

The built-in SystemC simulator is binary compatible with the OSCI SystemC 2.0.1 simulator. That means that you can link the object files (`*.{o,a,so}`) compiled with the OSCI SystemC 2.0.1 simulator to a `simv` executable.

The `-sysc=<version>` option accepts 2.0.1 as its valid argument. Any value other than this is considered invalid and in that case, VCS displays the following error message:

```
syscan timer.cpp:timer -sysc=2.2
Error: SystemC version 2.2 is not supported.
```

If you use the option `-sysc=<version>`, you must use it consistently during both analysis and compilation. Any mismatch in the version displays the error message:

```
Error: Mixing different SystemC versions is not allowed.  
      Either add or omit argument -sysc=<version> to all  
      calls of vlogan, vhdlan, syscan, and vcs  
      where you use SystemC.
```

Please note, this option is not a replacement for `-sc_model`.

Compiling Source Files

If you need to compile the source files, which include `systemc.h`, in your own environment and not with the `syscan` script, then add compiler flag `-I$VCS_HOME/include/systemc201` or `-I$VCS_HOME/include/systemc21` for SystemC2.0.1 and for SystemC2.1 respectively.

20

Using OpenVera Assertions

This chapter introduces the OpenVera Assertions (OVA) language and explains how to compile and run OVA within VCS. It covers the following topics:

- [Introducing OVA](#)
- [OVA Flow](#)
- [Checking OVA Code With the Linter Option](#)
- [Compiling Temporal Assertions Files](#)
- [OVA Runtime Options](#)
- [OpenVera Assertions Post-Processing](#)
- [Viewing Output Results](#)
- [Using OVA with Third Party Simulators](#)
- [Inlining OVA in Verilog](#)

- [Using Verilog Parameters in OVA Bind Statements](#)
- [OVA System Tasks and Functions](#)

For a detailed description of OpenVera Assertions, see the OpenVera Assertions Language Reference Manual.

Introducing OVA

OVA is a clear, easy way to describe sequences of events and facilities to test for their occurrence. With clear definitions and less code, testbench design is faster and easier, and you can be confident that you are testing the right sequences in the right way.

As a declarative method, OVA is much more concise and easier to read than the procedural descriptions provided by hardware description languages such as Verilog. With OVA:

- Descriptions can range from the most simple to the most complex logical and conditional combinations.
- Sequences can specify precise timing or a range of times.
- Descriptions can be associated with specified modules and module instances.
- Descriptions can be grouped as a library for repeated use. OVA includes a Checker Library of commonly used descriptions.

Built-in Test Facilities and Functions

OVA has built-in test facilities to minimize the amount of code that you need to write. In addition, OVA works seamlessly with other Synopsys tools to form a complete verification environment.

OVA performs the following tasks:

- Tests Verilog, VHDL, and mixed-HDL designs using VCS and VCS MX.
- Automatically tests and reports results on all defined sequences. You just write the definitions.
- Produces results that can be viewed with DVE.
- Can be monitored and controlled as part of a Vera testbench.

VCS also has functional coverage that provides you with code coverage information about your OVA code.

Using OVA Directives

OVA uses two directives:

- The `assert` directive, consists of mostly temporal expressions and is used to define a property of a system that is monitored to provide the user with a functional validation capability. Properties are specified as temporal expressions, where complex timing and functional relationships between values and events of the system are expressed.
- The `cover` directive consists of event coverage expressions used to record all successful matches of the coverage expression. When the event expression results in a match, the cover always increments a counter. Multiple matches per attempt may be generated and reported. With compile-time option `-ova_enable_diag`, if the match is the first success of the attempt, then the cover directive also increments a second counter `first_matches`.

How Sequences Are Tested Using the assert Directive

Testing starts with a *temporal assertion file*, which contains the descriptions of the sequences and instructions for how they should be tested. OVA is designed to resemble Verilog with similar data types, operators, and lexical conventions.

A typical temporal assertion file consists mostly of temporal expressions in OVA units. These temporal expressions are the descriptions of the event sequences. Events are values or changes in value of any OVA signal. Temporal expressions can be combined to form longer or more complex expressions. The language supports not only linear sequences but logical and conditional combinations.

When you instantiate the OVA unit, you connect the OVA signals to Verilog variables and nets. You cannot connect OVA signals to Verilog named events. The basic instruction for testing is a *temporal assertion*. Assertions specify an expression or combination of expressions to be tested. Assertions come in two forms: *check*, which succeeds when the simulation matches the expression, and *forbid*, which succeeds when the simulation does not match.

The temporal expressions and assertions must also be associated with a clock that specifies when the assertions are to be tested. Different assertions can be associated with different clocks. A clock can be defined as posedge, negedge, or any edge of a signal; or based on a temporal expression. Also, asynchronous events can use the simulation time as a clock.

An assertion can be associated with all instances of a specified module or limited to a specific instance.

Example 20-1 shows an example temporal assertion file. It tests for a simple sequence of values (4, 6, 9, 3) on the device's output bus.

Example 20-1 Temporal Assertion File, cnt.ova

```
/* Define a unit with expressions and assertions (or select
one from the Checker Library).
*/
unit step4
  #(parameter integer s0 = 0)          // Define parameters
  (logic clk, logic [7:0] result); // Define ports

  // Define a clock to synchronize attempts:
  clock posedge (clk)
  {
    // Define expressions:
    event t_0 : (result == s0);
    event t_1 : (result == 6);
    event t_2 : (result == 9);
    event t_3 : (result == 3);
    event t_normal_s: t_0 #1 t_1 #1 t_2 #1 t_3;
  }

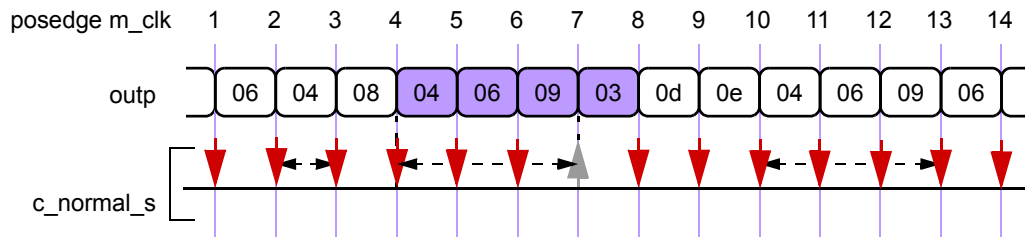
  // Define an assertion:
  assert c_normal_s : check(t_normal_s, "Missed a step.");

endunit

/* Bind the unit to one or more instances in the design.
*/
// bind module cnt : // All instances of cnt or
bind instances cnt_top.dut : // one instance.
  step4 start_4 // Name the unit instance.
  #(4) // Specify parameters.
  (m_clk, outp); // Specify ports.
```

When the temporal assertion file is compiled and run, the assertions are continuously tested for the duration of the simulation. New attempts to match each assertion to the simulation's values are started with every cycle of the assertion's associated clock. Each attempt continues until it either fails to match or succeeds in matching the complete expression (see Figure 20-1). The up arrow at clock tick 7 indicates a match that started at tick 4. The down arrows are failures. The failure or success of each attempt is logged to a file that you can review later.

Figure 20-1 Assertion Attempts for cnt.ova



Important:

Synopsys recommends always specifying a clock signal for an assertion. An assertion without a clock is called a simtime assertion. VCS checks simtime assertions with a default clock that is the equivalent to the smallest time precision in the design and this significantly impedes simulation performance. When VCS compiles a simtime assertion, it displays a warning message.

A OpenVera testbench can monitor and control the testing. Using built-in object classes, you can stop and start attempts to match the selected assertion; monitor attempts, failures, and successes; and synchronize the testbench with the testing process.

How Event Coverage Is Tested Using the cover Directive

The `cover` directive records only successful matches. You can specify the cover directive specific to your design or use it with assertions statements.

With the default compile-time options, only one counter is generated for each cover directive. This counter is incremented each time the event expression matches. At the end of a default simulation, the number of total matches is reported in the example:

```
unit_instance_name cover_name, int_val total match
```

You can also increment a second counter using the compile-time option `ova_enable_dialog`. In this case, the first counter is the same as the default counter, and the second counter reports the number of total matches of the event expression.

OVA Flow

OVA uses the following flow:

1. Create a temporal assertion or cover file. See the OpenVera Assertions Language Reference Manual (`$VCS_HOME/doc/UserGuide/ova_lrm.pdf`).

Start with simple temporal expressions and then combine them to form complex sequences. Simple expressions compile and run faster, and might use less memory.

Files named with an `.ova` extension (*filename.ova*) are recognized as assertion and cover files.

2. Compile and simulate the design, including Temporal Assertions files and options, on the `vcs` and `simv` command lines.
3. After running the simulation, verify the results:
 - See [“Viewing Results in a Report File”](#) on page 20-41.
 - See [“Viewing Results with Functional Coverage”](#) on page 20-42.
 - Results can also be monitored through a Vera testbench.

Checking OVA Code With the Linter Option

The linter option adds predefined rules. Rule violations other than syntax / semantic errors are not errors in the strict sense of the word, but are warnings of potential performance issues or differences in intended and real semantics of the OVA expressions.

The linter option has two sets of rules:

- General Rules (GR) that are applicable to both simulation and formal verification tools, synthesis tools such as VCS and verification tools
- Magellan Rules (MR) that are specific to Magellan (or other similar formal verification tools).

Upon detecting a violation of any one of the rules and normal parsing errors, the linter will output a message in the same format as the OVA parser does now. It will prefix the message by ERROR, WARNING or RECOMMENDATION, depending on the type of message.

The rules as listed next are classified as "e" for ERROR, "w" for WARNING, and "r" for RECOMMENDATION.

Each rule contains two information blocks. The first one is the error message text output by the linter. The second provides further information on the kind of problem identified.

Applying General Rules with VCS

The linter option in VCS is `-ova_lint`.

If used without any Verilog design file, only the OVA files are analyzed (including bind statements), and potential problems are reported. The command is:

```
vcs ova_files_list -ova_lint
```

If used with Verilog and OVA files (or inlined OVA), the compilation proceeds normally while detailed linting analysis is also done. If no fatal error is found, the simulation can go ahead and any recommendations from the linter can be incorporated for later runs.

Linters General Rule Messages

This section lists the messages generated by the general rules and describes the condition that caused the message along with a suggested remedy.

GR1: WARNING "assert forbid" used on an event that contains an "if" without an "else".

Example:

```
event e: if a then #1 b;  
assert c: forbid(e);
```

Whenever "a" is false, the assertion will fail because the "if - then" implication is satisfied. This may not be, however, what is intended. A modification to consider is to change the event definition as follows:

```
event e: a #1 b;
```

GR2: WARNING "ended" or "matched" is used on an event that contains an "if" without an "else"

Example:

```
event e1: if a then #1 b;  
event e2: if ended e1 then #1 c;
```

Whenever "a" is false, event "e1" will match because the "if - then" implication is satisfied. This means that "e2" will also trigger and try to match on "c" at the next clock tick. This may not be, however, what is intended. A modification to consider is to change event "e1" as follows:

```
event e1: a #1 b;
```

GR3: WARNING "if" appears in the middle of a longer sequence, or a composition of sequences where both contain an "if" without an "else"

Example:

```
event ev: if a then #1 (if b then #1 c);
```

If "a" is true, then in the next cycle event e will match even if "b" is false. This may not be the intended behavior.

Consider changing the event as follows:

```
event ev: if a then (#1 b #1 c);
```

Note 1: The portion of the original event "ev" in parentheses could have been an event declared separately and instantiated in "e", thus possibly hiding the fact that it contains an "if".

Note 2: The following use of if-then-else can be useful, however:

```
event ev: if a then #1 if b then d else e;
```

Here a, b, c, d and e are some boolean expressions.

GR6: WARNING OVA * repetition factor contains a 0.

Example:

```
a*[m..n] #k b;
```

Here m is a 0 or a parameter with default value of 0

Consider rephrasing the expression using for instance a disjunction, e.g., if m is 0 then (a*[1..n] #k b) || (#(k-1) b);

GR8: WARNING "matched" used in the same clock domain (one clock tick delay)

Example:

```
clock posedge clk {
    event e1: ... ;
    event e2: if matched e1 then ... ;
}
```

The successful match on "e1" would only be detected in "e2" at the subsequent posedge of "clk". Consider changing "e2" as follows:

```
event e2: if ended e1 then ... ;
```

The "ended" operator transfers the match of "e2" to "e2" at the same "posedge clk".

GR11: RECOMMENDATION event contains a large delay or repetition interval.

Example:

```
event e1: if posedge a then a*[1..1000] #1 b;
```

or

```
event e2: a #[1..10000] b;
```

or

```
event e3: a #10000 b;
```

Consider using a variable to count clock ticks if there are no overlapped transactions. For example, the first case:

```
logic [9:0] cnt = 11'b0;
clock ... {
    cnt <= reset ? 10'd0 :
```

```

        posedge a ? 10'd1 :
            cnt > 10'd1000 ? cnt :
                cnt = cnt + 10'd1;
event e1: if posedge a then ((cnt <= 11'd1000) && a) * [1..] #1 b;
}

```

Alternately, if overlaps are possible consider using time stamps stored in a queue (see for instance the OVA standard checker unit "ova_req_ack_unique").

GR14: RECOMMENDATION top-level conjunctions over events in "check" assertions.

Example:

```

clock posedge clk {
    event e1: ...;
    ...
    event eN: ...;
    event e: if a then e1 && e2 && e3 && ... ;
}
assert c: check(e);

```

Consider placing an assertion on each individual event as follows:

```

assert c1: check (if a then e1);
assert c2: check (if a then e2);
assert c3: check (if a then e3);
...

```

This creates more assertions but they may execute faster because they can be considerably simpler than event "e".

GR15: RECOMMENDATION Simulation time is used as the sampling clock

The event used in an assertion does not have a sampling clock. The simulation time will be used in that case which may lead to inefficient simulation. Consider whether such fine sampling is required in your application.

GR17: RECOMMENDATION for loop over events / assertions with large (>20) ranges with loop index appearing in delay or repetition operators.

Consider using variables and / or compacting into a single event. For an example see the OVA standard checker "ova_req_ack_unique". It contains two versions of a checker, version 0 that is suitable for small ranges of parameters, and version 1 that uses a time stamp and a queue when the loop ranges become large (thus creating too many events / assertions).

GR19: WARNING the "past" operator is over a long time interval (> 1000).

Consider if the same property can be expressed differently without the use of a deep look into the past (or in the future using # delay, which would have a similar problem.)

GR25: WARNING a conditional check assertion that involves open-ended delay interval in the consequent (unbounded eventuality).

Example:

```
event e: if c then some_sequence1 #[1..] some_sequence2;
```

This assertion cannot fail in a simulation because of the failure of "some_sequence2" because "some_sequence2" could occur after the simulation ends.

Consider placing an upper bound on the delay interval as follows:

```
event e: if c then some_sequence1 #[1..20] some_sequence2;
```

GR26: WARNING OVA variable not initialized to a known value.

Consider initializing the variable to 0. For example:

```
logic v = 1'b0;
```

This may be needed if the OVA checker is to be used with formal tools.

GR27: WARNING assert check over a non-bool sequence that does not start with an "if".

Example:

```
event e: a #1 b;  
assert c: check(e);
```

The assertion will fail any time "a" is false. Unless used as a coverage assertion that is supposed to track the occurrences of the sequence "a" followed by "b" in the simulation trace, or when the sampling clock is some irregular event from the design, the usefulness of the assertion as a checker should be reconsidered, as it would require "a" to hold at every clock tick in order to have even a chance to succeed.

GR31: RECOMMENDATION multiple attempts may be triggered for the same check.

Example:

Suppose that "req" must return to 0 between activations, and if asserted it must remain so until "ack" is received, then the following sequence and an assertion on it would create unnecessary additional attempts to be triggered for the same req-ack transaction:

```
event e: if req then req*[1..] #0 ack;
```

Consider replacing it with:

```
event e: if posedge req then req*[1..] #0 ack;
```

The modified event will generate only one non-vacuous attempt for each assertion of "req".

GR32: WARNING the unit instance name in a bind statement is missing.

The problem is that an automatically generated name is inserted by the compiler which makes it more difficult for the user to locate the checker instance.

GR34: WARNING multiplication *N (by a constant) is the last operator on an expression - consider changing to repetition *[N].

Example:

```
event e: a #[1..] b*3;
```

Often [] is omitted by accident from the repetition operation. Consider changing the expression to event e:

```
a #[1..] b*[3];
```

GR35: WARNING a bitwise operator (&, |, ~, etc.) is used on a boolean expression.

Example:

```
logic a; logic b;  
event e: (a & b) == 0 #1 (~b == 0);
```

The problem is that with bitwise operations the word extension to 32 bits as implied by the "0" operand and the bitwise operations may produce unwanted results. Consider rewriting as follows:

```
event e: (a && b) == 0 #1 (!b == 0);
```

GR36: RECOMMENDATION open-ended interval delay used in an event to which ended or matched is applied.

Example:

```
event e1: a #[1..] b;  
event e2: ended e1 #1 ... ;
```

The problem is that any evaluation attempt that matches on "a" will remain active till the end of simulation, waiting for (yet another) occurrence of "b". Most likely this is not intended. Consider rewriting "e1" to terminate on the first match of "b" as follows:

```
event e1: a #1 (!b)*[0..] #1 b;
```

Applying Magellan Rules for Formal Verification

This section describes use of Magellan Rules (MR) for checking OVA code to be used with formal verification tools such as Magellan.

The compile-time option for enabling MR rules is

```
-ova_lint_magellan.
```

Linters General Rule Messages:

This section lists the messages generated by the general rules and describes the condition that caused the message along with a recommended alternate model.

MR1: ERROR the unit instance name in a bind statement is missing. The instance name must be provided.

MR2: WARNING an assertion is stated solely over an OVA variable value.

Example:

```
logic [bw-1:0] tmp = 1'b0;
clock posedge clk {
    tmp <= c1 ? reg_A :
           c2 ? reg_B;
    event e: |tmp == 1'b1;
}
assert c: check(tmp);
```

This type of an assertion cannot be used effectively as an assumption in Magellan because it requires constraining the signals driving the OVA variable "tmp" in the past clock tick. This is not possible when doing random simulation constrained by OVA assertions. The result is that no constraint is imposed. Consider correcting the assertion as follows:

```
logic [bw-1:0] tmp;
assign tmp = c1 ? reg_A :
            c2 ? reg_B;
    clock posedge clk {
        event e: |tmp == 1'b1;
    }
assert c: check(tmp);
```

The constraint is now applied in the current clock cycle rather than in the past one.

MR3: WARNING the "matched" or "ended" operator appears in the consequent sequence of a "check" assertion or as part of the sequence in a "forbid" assertion.

The problem is that this form of an assertion cannot be used effectively in random simulation under OVA constraints / assumption because it requires constraining inputs in past clock cycles.

Example:

```
clock posedge clk {
    event e1: c #1 d;
    event e2: if a then matched e1;
}
assert c: check(e2);
```

Try to rewrite event "e2" without the use of "matched". In the case of our simple example, the solution is simple due to the one cycle delay introduced by "matched":

```
clock posedge clk {
//      event e1: c #1 d; -- do not use
      event e2: if a then c #1 d;
}
assert c: check(e2);
```

In general the transformation may not be as simple as in the above example. It may be preferable to approach the problem differently right from the start rather than trying to rewrite the case later.

MR5: ERROR simulation time is used as the OVA clock.

The notion of simulation time is not available in Magellan. However, you can create an explicit periodic clock that provides some notion of time advancement.

MR6: ERROR case equality is used in expressions. This is non-synthesizable.

MR7: ERROR comparisons with 'z' and 'x' values is used. This is non-synthesizable.

MR8: WARNING an uninitialized OVA variable is used.

An uninitialized variable may cause a simulation - formal mismatch due to differences in interpreting the initial unknown value "x".

Compiling Temporal Assertions Files

Temporal assertions files are compiled concurrently with Verilog source files. You can use a set of OVA-specific compile-time options to control how VCS compiles the temporal assertions files.

Note:

When you use the OVA compile-time options, VCS creates a Verification Database directory in your current directory (by default named `simv.vdb`). VCS writes intermediate files and reports about OpenVera Assertions in subdirectories in this directory.

The following compile-time options are for OVA:

`-ovac`

Starts the OVA compiler for checking the syntax of OVA files that you specify on the `vcs` command line. This option is for when you first start writing OVA files and need to make sure that they can compile correctly.

`-ova_cov`

Enables viewing results with functional coverage.

`-ova_cov_events`

Enables coverage reporting of expressions.

`-ova_cov_hier filename`

Limits functional coverage to the module instances specified in *filename*. Specify the instances using the same format as VCS coverage metrics. If this option is not used, coverage is implemented on the whole design.

`-ova_debug`

Required to view results with DVE.

`-ova_dir pathname`

Specifies an alternative name and location for the Verification Database directory. There is no need to specify the name and location of the new Verification Database directory at runtime. The `simv` executable contains this information.

If you move or rename this directory after you create the `simv` executable, you include this option at runtime to tell VCS its new name or location.

`-ova_file filename`

Identifies *filename* as an assertion file. Not required if the file name ends with `.ova`. For multiple assertion files, repeat this option with each file.

`-ova_filter_past`

For assertions that are defined with the `past` operator, ignore these assertions where the past history buffer is empty. For instance, at the very beginning of the simulation the past history buffer is empty. So, a check/forbid at the first sampling point and subsequent sampling points should be ignored until the past buffer has been filled with respect to the sampling point.

`-ova_enable_diag`

Enables further control of result reporting with runtime options.

Used with the `cover` directive, `ova_enable_diag` generates a second counter to report the number of times an attempt to match the event expression succeeds for the first time.

The counters are reported in the form

```
unit_instance_name cover_name,  int_val  total match,  
int_val  first match
```

For example it can be:

```
top.\gen_2.aova2 .cover_temp_no_vacuous_f_eq_t, 4 total  
match, 4 first match
```

`-ova_inline`

Enables compiling of OVA code that is written inline with a Verilog design.

Note:

You can also use the VCS `-f` option to specify a file containing a list of all absolute pathnames for Verilog source files and compile-time option. You can pass all OVA compile-time options through this file, except `-ova_debug`.

The `-assert` compile-time option and keyword arguments, that were implemented for SystemVerilog assertions, also work on OpenVera assertions. The `filter_past` keyword argument, for the `$past` system function, also works for the `past` operator. See ["Options for Compiling OpenVera Assertions \(OVA\)" in Appendix B](#) and ["Options for Simulating OpenVera Assertions" in Appendix C](#).

OVA Runtime Options

The following runtime options are available for use with OVA:

`-ova_quiet [1]`

Disables printing results on screen. The report file is not affected. With the 1 argument, only a summary is printed on screen.

`-ova_report [filename]`
Generates a report file in addition to printing results on your screen. Specifying the full path name of the report file overrides the default report name and location, which is `./simv.vdb/report/ova.report`.

`-ova_verbose`
Adds more information to the end of the report including assertions that never triggered and attempts that did not finish, and a summary with the number of assertions present, attempted, and failed.

A set of runtime options are also available for controlling how VCS writes its report on OpenVera Assertions. You can use these options only if you compiled with the `-ova_enable_diag` compile-time option.

`-ova_filter`
Blocks reporting of trivial if-then successes. These happen when an if-then construct registers a success only because the if portion is false (and so the then portion is not checked). With this option, reporting only shows successes in which the whole expression matched.

`-ova_max_fail N`
Limits the number of failures for each assertion to *N*. When the limit is reached, the assertion is disabled. *N* must be supplied, otherwise no limit is set.

`-ova_max_success N`
Limits the total number of reported successes to *N*. *N* must be supplied, otherwise no limit is set. The monitoring of assertions continues, even after the limit is reached.

`-ova_simend_max_fail N`
Terminates the simulation if the number of failures for any assertion reaches *N*. *N* must be supplied, otherwise no limit is set.

`-ova_success`
Enables reporting of successful assertion matches in addition to failures. The default is to report only failures.

For a `cover` statement, triggers the match (success) message in the following format:

```
Ova [i][j]: ".fileName", <line>: <hierCoverName> started  
at 900s covered at 900s [optional custom msg]
```

Here *i* and *j* are the severity and category values associated with the cover statement, respectively. `ova_success` is under the control of `-ova_quiet`.

The `-assert` runtime option and keyword arguments, that were implemented for SystemVerilog assertions, also work on OpenVera assertions. For example, the `-assert nocovdb` runtime option and keyword argument tells VCS not to write the OpenVera assertions database file as well as not to write the SystemVerilog assertions database file. See ["Options for Compiling OpenVera Assertions \(OVA\)" in Appendix B](#) and ["Options for Simulating OpenVera Assertions" in Appendix C](#).

Functional Code Coverage Options

Functional coverage is code coverage for your OVA code. With functional coverage enabled, the `cover` statement is treated in the same manner as an `assert` statement. The runtime options are enabled by the `-ova_cov` compile-time option. These runtime options are as follows:

`-ova_cov`

Enables functional coverage reporting.

`-ova_cov_name filename`

Specifies the file name or the full path name of the functional coverage report file. This option overrides the default report name and location. If only a file name is given, the default location is used resulting in: `./simv.vdb/fcov/filename.db`.

`-ova_cov_db path/filename`

Specifies the path and filename 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/fcov`.

OpenVera Assertions Post-Processing

You can use VCS to build a post processor from a compiled design and temporal assertion files. You then run the post-processor, using either DVE or the post-processor CLI supplied with OVAPP, as you would a simulation compiled with the design and OVA files together. This approach allows you to:

- Post-process a compiled design several times with different temporal assertions files each time. You can observe the effects of different assertion scenarios and collect several distinct sets of functional coverage data.
- Develop assertions incrementally over a series of post-processing runs, improving and augmenting the assertions in the process.

OVAPP Flow

The following steps show a typical flow for post-processing a compiled VCS design with temporal assertions.

1. To use the post-processor CLI as the debugging tool, include the `$vcdpluson` or the `$dumpvars` system task in your Verilog code.
2. Compile your design in VCS with the `-ova_PP` compile-time option.

Note:

Use the `$vcdpluson` or the `$dumpvars` system task in your Verilog code to create dump files and enable CLI functionality. Do not use the `-ova_debug` compile-time option.

3. Simulate the design to create a VPD or VCD file.
4. Build the post-processor.
5. Run the post-processor using DVE or the post-processor CLI.

Building and Running a Post-Processor

The procedure to build and run a post-processor is as follows:

1. Include either of the following in your Verilog code:
 - The `$vcdpluson` system task to tell VCS to write a VPD file during simulation.

Note: Use the `-PP` compile-time option to write the VPD files as described in the following step.
 - The `$dumpvars` system task to tell VCS to write a VCD file during simulation.

If VCS writes a VCD file, the post-processor will call the `vcd2vpd` utility to translate it into a VPD file.

Note:

Do not use the `-ova_PP` compile-time option to generate design dumps for OVAPP. Such dumps will not contain all the correct hierarchies needed.

2. Compile your design with the `-ova_PP` compile-time option, for example:

```
vcs -f filename -ova_PP [-PP] [-o simv_name]  
[-ova_dir directory_path]
```

In this example compilation command line:

`-f filename`

Specifies a file containing the source files, and perhaps compile-time options. This compile-time option is not specifically related to OVA post-processing.

`-ova_PP`

Tells VCS to record in the verification database directory design information that is used by the post-processor.

By default VCS also creates the `simv.vdb` directory in the current directory when you include this option. We call this directory the verification database directory. VCS writes post-processing data in a subdirectory in the verification database directory

`-PP`

Optional compile-time option that enables VCS to write a VPD file. This option also enables VCS to recognize the `$vcdpluson` system task. This compile-time option is not specifically related to OVA post-processing.

`-o simv_name`

This compile-time option is not specifically related to OVA post-processing. It specifies an alternative name, and possibly a different location, for the `simv` executable. Because the executable and this directory have the same name (but not the same extension), you will alter the name and location of the verification database directory.

`-ova_dir directory_path`

Specifies an alternative name and location for the verification database directory. This option supersedes the `-o` option in naming and locating the verification database directory.

During compilation VCS displays the following message:

```
Generating OVA post-processing data ...
```

3. Simulate the design. There are no runtime options needed for the post-processor. As usual, if you used the `-o` compile-time option to specify the name of the `simv` executable, you enter this name, instead of `simv`, to start the command line.

During simulation VCS writes the VPD file.

4. The next step is to build the post-processor from the post-processing data and one or more temporal assertions files. You specify building the post-processor and the temporal assertions file with the `-ova_RPP` compile-time option. For example:

```
vcs -ova_RPP filename.ova...[-o simv_name] [-ova_dir
directory_path] [-ova_cov]
```

In this example compilation command line:

`-ova_RPP`

Tells VCS to compile the post-processing engine. VCS writes it in a subdirectory in the verification database directory (VDB).

`filename.ova`

Temporal assertions file whose assertions you want compiled into the post-processing engine. You can specify more than one temporal assertions file.

`-o simv_name`

If you included the `-o` option when you compiled your design, also include it on this command line to tell VCS where to look for the VDB in which the new data generated by the current command/step will be written.

Note:

Including this option also creates the `simv_name.daidir` direct access interface directory. This directory enables you to use CLI commands during post-processing.

`-ova_dir directory_path`

Specifies the verification database directory that VCS searches for the information about your design that is used by the post-processor.

`-vdb_lib directory_path`

Specifies an alternative location for the VDB that VCS searches for data generated during a prior step. VCS first searches the directory specified by the `-ova_dir` option, then the directory specified by this option. If it does not find this information in either directory, the compilation fails. If you include this option without the `-ova_dir` option, VCS searches the directory specified by this option, then the `simv.vdb` directory in the current directory.

`-ova_cov`

Enables the post-processor to gather OVA functional coverage information.

5. The last step is to start the post-processor. You do this with an `ovapp` command line. Its syntax is as follows:

```
ovapp [-vdb_lib directory_path] [-vpd filename.vpd]
[-vcd filename.vcd] [-cli [-daidir=pathname.daidir] |
-ova_report [filename] -ova_cov]
[-ova_name session_name] [-o simv_name]
[other_OVA_options]
```

In this example:

`-vdb_lib directory_path`

Specifies the verification database directory that contains the dynamic library, the post-processing engine.

`-vpd filename.vpd`

Specifies the VPD file. If the filename is `vcdplus.vpd` and it is in the current directory, you can omit this option.

- vcd filename.vcd
Specifies the VCD file. The post-processor will call the vcd2vpd utility to translate it to a VPD file name vcdplus.vpd. If the VCD file is named verilog.dump and it is in the current directory (if the current directory doesn't contain a file named vcdplus.vpd). You can omit this option and the post-processor will use the verilog.dump file.
- cli
Specifies that post-processing starts with a command line prompt for entering CLI commands. These CLI commands are the same as the VCS CLI commands plus additional ones for OVA post-processing. See [“OVA Post-Processing CLI Commands” on page -20-31](#).
- daidir=pathname.daidir
Specifies the direct access interface directory used by the post-processor for CLI commands.
- ova_report [filename]
Generates a report file in addition to printing results on screen. Specifying the full path name of the report file overrides the default report name and location, which is simv.vdb/report/ova.report-ova_cov. Tells the post-processor to also gather functional OVA coverage data.
- ova_name session_name
Changes the name, but not the extension, of the generated files in the verification database directory. Generated files are those specified by other options such as -ova_report. See [“Using Multiple Post-Processing Sessions” on page -20-32](#).

`-o simv_name`

Specifies the executable name so the post-processor knows the name and location of the post-processing engine and the verification database directory. If you used the `-o compile-time` option when you built the post-processor, you need to enter it again on this command line so that the post-processor will know the name and location of the dynamic library.

OVA Post-Processing CLI Commands

When you include the `-cli` option, the post-processor displays a CLI command prompt just like the VCS CLI command prompt:

```
cli_0>
```

You can enter any VCS CLI command at this prompt, such as those for moving up and down the hierarchy and displaying values. There are also special CLI commands for OVA post-processing. The special CLI commands are as follows:

```
pp_fastforward time
```

Advances post-processing to the specified simulation time.

```
pp_rewind time
```

Returns post-processing to the specified previous simulation time.

```
ova_trace_off instance_hierarchical_name assertion_name time
```

Disables the tracing of the specified assertion in the specified instance, at the specified simulation time.

```
ova_trace_off assertion_hierarchical_name
```

Disables tracing of the specified assertion the next time.

```
ova_trace_on instance_hierarchical_name assertion_name time
```

Enables the tracing of the specified assertion in the specified instance, at the specified simulation time.

```
ova_trace_on assertion_hierarchical_name  
    Enables tracing of the specified assertion name the next time.
```

Using Multiple Post-Processing Sessions

You can repeatedly run the post-processor using different input (either different temporal assertion files or different waveforms). This section describes how to use the `-ova_name` option to generate a unique report for each session. For example:

```
vcs -ova_RPP first.ova  
  
ovapp -ova_name first -ova_report  
  
vcs -ova_RPP second.ova  
  
ovapp -ova_name second -ova_report
```

After these two post-processing sessions, the `simv.vdb/report` directory contains:

```
first.report    second.report
```

Multiple OVA Post-Processing Sessions in One Directory

You can run multiple OVA post-process sessions in the same directory. While only one design should be simulated in any one directory, any number of OVA assertion sets and any number of stimulus waveform patterns can be executed against that one design from within the same working directory.

In this section, reference will be made to the following four-step post-process flow:

1. Capture waveforms from a simulation of the base design.
2. Compile the base design for post-processing (`vcs -ova_PP`).
3. Compile the OVA checkers against the base design (`vcs -ova_RPP`).
4. Replay the captured waveforms against the compiled checkers (`ovapp`).

Note that step 1 could also be run after step 2 using the executable generated as part of the compilation of the base design. Note also that step 3 could be run more than once with different OVA checkers and that step 4 could be run more than once with different captured waveform files as input.

In each of these steps, information generated by one step is read and used by the immediately succeeding step. In step 3, the skeleton design data generated in step 2 is used to compile the OVA checkers into an engine that will be used during the post-processing replay. In step 4, the compiled OVA runtime engine and checker database are used during the replay. All this intermediate data is stored in predetermined locations in the vdb directory.

You can select two separate vdb directories via the command line at each step in the flow. One of these directories takes the same basename as the simulation executable (derived from the `-o` option). This is the vdb directory into which the results of the current step are written.

Note:

In this discussion, the vdb directory derived from the basename given with the `-o` option will be referred to as `simv.vdb` for simplicity.

The other directory is a read-only directory that you can specify with the `-vdb_lib` option. This directory is used as the source for files generated by the previous step, but the files are not located in the `simv.vdb` directory (as specified by the `-o` option). If no `-vdb_lib` option is given, all intermediate files are expected to be found in the `simv.vdb` directory (the simple/default case).

To run multiple independent versions of either step 3 or step 4, each command must specify a directory into which it will write its results. Specify the directories with the `-o basename` option and argument (the OVA intermediate files, for example, will be written to a directory with the given *basename* plus the `.vdb` extension). The *basename* given for each unique run of each step must be unique.

Each command will also include a pointer to the vdb directory of the previous step against which this step is being performed. This path is specified with the `-vdb_lib` option. The arguments for both `-o` and `-vdb_lib` can include full path names if desired.

Here is an example of the overall flow:

1. `vcs -ova_PP -o simv1 verilog_files`
2. `vcs -ova_RPP -vdb_lib simv1.vdb -o simv2 ova_files`
3. `ovapp -vdb_lib simv2.vdb -o simv3 -vcd dumpfile`

As you can see, in step1, we pass `simv1` as the basename for the intermediate output files. In step 2, we pass the `vdb` directory created as a result of step1 as the argument of the `-vdb_lib` option and give VCS a second basename, `simv2`, for the new intermediate file directory. When VCS tries to locate a file in the `simv2.vdb` directory (during step 2) and cannot find the file, it next checks the directory specified by the `-vdb_lib` option. This way, you can run many unique step 2 OVAPP compilations using the same base design without running the risk of overwriting necessary files generated by the base design compilation.

In addition to the `-o` and `-vdb_lib` options, other compile-time options must be taken into consideration when running multiple compilations, simulations, or post-process runs in the same directory. The following sections describe in detail some of the issues to consider.

Interactive Simulation

The initial compile can be used as an interactive/batch simulation, with or without OVA checking. Simply add `-PP` or `-I` to the step 1 compile command line.

Waveform Dump Files

The post-processing run requires a waveform dump file. This file can be either a VCD file or a VPD file. The file can be generated by any simulation which is based on the same design as was compiled in step1 (the `-ova_PP` step). The dump file can be generated by the executable resulting from the Step1 compile, an earlier interactive simulation of the same design, or some other simulation.

This dump file must contain, as a minimum, any signal referenced by the OVA checkers to be post-processed. The signals used by the OVA checkers must be included in a `$dumpvars` or `$vcdpluson` dumped scope (an instance specified in one of these system tasks) in order to be visible by the post-process checkers. Automated dumping of OVA signals is not supported.

The runtime option `-vcd filename` can be used to name the VCD file if there is a `$dumpvars` system task in the design. The `+vpdfile filename` option and argument can be used to name the VPD file if there is a `$vcdpluson` system task in the design.

If you include the signals referenced by the checkers in `$vcdpluson` and `-PP` or `$dumpvars`, the referenced signals will be dumped at compile time.

Note:

Do not use `-ova_debug` to generate dump files, since the generated files will not contain all the right hierarchies.

It is recommended that the dump files be explicitly named in all cases. Both the `simv` executable and the `ovapp` executable use the same default name for the dumpfile output. If you dump a VPD file by its default name in step 1 and use this file as the input to `ovapp`, it is possible that the input dumpfile can be overwritten.

Note that the step 3 (`ovapp`) waveform input file is specified with the `-vcd` or `-vpd` options. The `-vcd vcdfile` option tells `ovapp` to read a VCD file, while `-vpd vpdfile` tells `ovapp` to read a VPD file. This `-vcd` option is NOT the same option used by VCS to name the output waveform file. Caution in using this option is recommended.

Note that OVAPP internally requires a VPD file. Therefore, if you pass a VCD file to OVAPP, it calls the vcd2vpd converter prior to the post-processing run. This means that using VPD files to capture the waveforms to be replayed will result in better overall performance.

Dumping Signals Automatically

If you include the OVA checkers to the compilation step before the waveform replay dump is generated, these signals will be included in the dump automatically.

Debugging

When the OVA files are compiled in step 2 (`-ova_RPP`), debugging is enabled by default, and the `-ova_enable_diag` and `+cli` compile-time options are entered automatically. This is to support the OVAPP debugger.

PLI or Other 3rd Party Tools

If the initial compilation includes PLI or other 3rd party interfaces that use PLI/DKI to interface to VCS, a daidir directory will be generated during the step1 compile. To keep from corrupting this directory, the `-o` option for the step 2 compile **must** name a different basename from that used for the step1 compile. For safety, it's best to always include the `-o` option and use a unique basename for each compilation.

Incremental Compile

The step 2 OVA engine compile (`-ova_RPP`) uses incremental compile in order to speed up the compilation process for multiple runs. Incremental compile generates intermediate files in a directory named `csrc` by default. If you enable the incremental compile feature in the step1 compile, it is possible for the `csrc` intermediate files to be corrupted. If you use incremental compile, you should also add the `-Mdir=dirname` option to the command line to direct the VCScompiler to store its intermediate files in a unique directory.

Inlined OVA

The step1 compile command can include the `-ova_inline` option to enable the capture and processing of OVA statements inlined as pragmas in the Verilog source. These pragmas are not copied to the skeleton design and thus will not be processed during the post-processing playback. This is in keeping with the strategy of post-processing, which allows playback of one or more sets of signal waveforms against one or more sets of OVA checkers. Post-processing was designed to playback against checkers defined in independent standalone OVA files (those given on the step 2 (`-ova_RPP`) command line).

Inlined OVA statements will not, however, interfere with the post-processing execution of the OVA statements compiled in step 2. Inlined OVA may be freely included in the design for interactive/ debugging purposes, and inlined statements will simply be excluded from the post-processing runs.

Reporting

By default, the assertion report generated by the OVA engine is written into `./simv.vdb/report/ova.report`. In the case of multiple post-processing runs, there is a chance the file will be overwritten. For each run, it is suggested that the `-ova_report name.report` and `-ova_name name` options be used to ensure that any report files generated will be stored under unique names.

Coverage

To enable functional coverage, use the `-ova_cov` option during the Step 2 (`-ova_RPP`) compile (also enter `-ova_cov_events` to see coverage of events). During the post-processing run, the `-ova_cov` option must again be given (as a runtime option) to actually turn on coverage capture.

By default, the coverage from all post-processing runs with a given compiled OVA image is captured in a single database. If you need to generate reports for each post-process run separately, use the `-ova_cov_name name` option to assign a unique name to each post-processing run. The various databases are stored under the `simv.vdb` directory in either case. Coverage reporting can include a single post-processing run or a merged set of runs, as described in the OVA chapter of the *VCS/VCSi User Guide*.

Coverage reports are generated with the `fcovReport` utility. The `fcovReport` command line should include the `-ova_cov_db vdbdir` option to point to the directory where the global coverage database resides, and the `-ova_cov_report name` to point to the path and name of the report file.

Things to watch out for

- If you pass *path/name* (a full path) to `-ova_cov_report`, `fcovReport` does not automatically create the directories in *path*. If the *path* does not exist, the report is not created and no error message is generated.
- The `-ova_inline` compile-time option should not be included with the `-ova_PP` compile-time option. If both options are present on the same command line, and inlined OVA references to checker library elements are included in the design, an error message will result. This error can be ignored and both the interactive simulation and the post-process compile should run fine.

Note:

Inlined OVA is not recognized for post-processing.

- If you use the `+vpdfile+filename` option to name the debug VPD file in the `ovapp` step, an informational message referring to a dummy file will be emitted. This message can be safely ignored.
- If the design from step 1 (`-ova_PP`) is recompiled, the step 2 OVA compilations might have to be rerun. If there are no structural changes to the design (no hierarchy changes and no added/deleted signals), it may not be necessary to recompile the OVA files. However, the step 1 design recompile should not be done while step 2 OVA compilations are running, because the design compile deletes and regenerates the skeleton design file, even if there are no changes to the actual design.

Viewing Output Results

The two main ways of viewing the results of a simulation involving OVA are:

- Viewing Results in a Report File
- Viewing Results with Functional Coverage

Viewing Results in a Report File

A report file is created when the `-ova_report` runtime option is used. The report file name is `ova.report` unless you specified a different name in the run command. This default `ova.report` file is stored in directory `simv.vdb/report`, where the `.vdb` directory is the root directory at the same level where the design is compiled and `simv` is stored.

To override this default name and location for the report file, use the `-ova_report` runtime option and provide the full path name of the report file.

The report file is replaced with each simulation run. If you want to save the results, be sure to rename the file before running the simulation again.

Assertion attempts generate messages with the following format:

Severity File and line with the assertion Full hierarchical name of the assertion Start time Status (succeeded at ..., failed at ..., not finished)

```
Ova [0]: "cnt.ova", 10: cnt.dut.c_normal_s: started at 5ns failed at 9ns,
"Wrong result sequence.",
  Offending 'outp == 4 #1 outp == 6 #1 outp == 9 #1 outp == 3'
```

Optional user-defined failure message Expression that failed (only with failure of check assertions)

Viewing Results with Functional Coverage

After running a series of simulations, you can generate a report summarizing the coverage of the assertions and expressions in the following two ways:

- With the default report, you can quickly see if all assertions were attempted, how often they were successful, and how often they failed. Potential problem areas can be easily identified. The report can cover one test or merge the results of a test suite. The report is presented in HTML and you can customize it with a Tcl script.
- An assertion and event summary report describes the total number of assertions and events details of their performance. This list can be filtered by category and severity to report matching assertions.

Using the Default Report

The default report shows the number of assertions and expressions that:

- Were attempted

- Had successes
- Had failures

Coverage is broken down by module and instance, showing the number of attempts, failures, and successes for each assertion and expression. Because if-then constructs register a success anytime the if portion is false (and so the then portion is not checked), the report also shows the number of *real successes* in which the whole expression matched. This works with nested if statements too.

Functional coverage can also grade the effectiveness of tests, producing a list of the minimum set of tests that meet the coverage target. Tests can be graded on any of these metrics:

- Number of successful assertion attempts versus number of assertions (*metric = SN*)
- Number of failed assertion attempts versus number of assertions (*metric = FN*)
- Number of assertion attempts versus number of assertions (*metric = AN*)
- Number of successful assertion attempts versus number of assertion attempts (*metric = SA*)
- Number of failed assertion attempts versus number of assertion attempts (*metric = FA*)

To generate a report, run the following command:

```
fcovReport [options]
```

Assertion and Event Summary Report

Since no changes are introduced to the data collection process when generating functional coverage reports, you can produce different reports from a simulation. One report could show all assertions and events; another report could show assertions filtered by category and severity.

The assertion and event summary report generates four html files:

- The report.index.html file displays total assertions and events and details including:
 - Assertions with at least 1 real success
 - Assertions with at least 1 failure
 - Assertions with at least 1 incomplete
 - Assertions without attempts
 - Events with at least 1 attempt
 - Events with at least 1 real match
 - Events without any match or with only vacuous matches
 - Events without any attempts

The report.index.html file also contains links to the other three files.

- The tests.html file describes the tests merged to generate the report.
- The hier.html file displays a hierarchical report table showing a list of instances, the number of assertions in each instance, and the number of events in each instance.

- The `category.html` file is generated when `-ova_cov_category` and/or `-ova_cov_severity` are used to filter results. Tables display functional coverage results for assertions showing the assertions having the category and severity specified along with number of attempts, successes, failures and incompletes.

To generate the assertion and event summary report, run the `fcovReport` command after compilation and simulation:

```
fcovReport [-ova_cov_severity value,...]
[-ova_cov_category value,...]
```

Command Line Options

The command line options are as follows:

`-e TCL_script | -`

Use this option to produce a custom report using Tcl scripts or entering Tcl commands at standard input (keyboard). Most of the other `fcovReport` options are processed before the Tcl scripts or keyboard entries. The exception is `-ova_cov_report`, which is ignored. Its function should be in the Tcl.

`-e TCL_script`

Specifies the path name of a Tcl script to execute. To use multiple scripts, repeat this option with each script's path name. They are processed in the order listed.

`-e -`

Specifies your intent to enter Tcl commands at the keyword.

You can input the Tcl commands provided by VCS to `fcovReport` for OVA coverage reports (see [“Tcl Commands For SVA And OVA Functional Coverage Reports” on page 23-49](#)), and to `assertCovReport` for SystemVerilog assertion (SVA) coverage.

- ova_cov_cover
Specifies reporting of cover directive information only.
- ova_cov_db *path*
Specifies the path of the template database. If this option is not included, fcovReport uses simv.vdb.
- ova_cov_events
Specifies reporting only about OVA events.
- ova_cov_grade_instances *target, metric*
[, *time_limit*]
Generates an additional report, grade.html, that lists the minimum set of tests that add up to the target value for the metric (see previous page for *metric* codes). The grading is by instance.
- ova_cov_grade_modules *target, metric*
[, *time_limit*]
Generates an additional report, grade.html, that lists the minimum set of tests that add up to the target value for the metric (see previous page for *metric* codes). The grading is by module.
- ova_cov_map *filename*
Maps the module instances of one design onto another while merging the results. For example, use this to merge the functional coverage results of unit tests with the results of system tests. Give the path name of a file that lists the hierarchical names of from/to pairs of instances with one pair per line:

from_name to_name

The results from the first instance are merged with the results of the second instance in the report.

`-ova_cov_merge filename`

Specifies the path name of a functional coverage result file or directory to be included in the report. If *filename* is a directory, all coverage result files under that directory are merged. Repeat this option for any result file or directory to be merged into this report. If this option is not used, fcovReport merges all the result files in the directory of the template database (specified with `-ova_cov_db` or *simv.vdb/fcov* by default).

`-ova_cov_report name | path/name`

Specifies the base name for the report. The fcovReport command creates an HTML index file at *simv.vdb/reports/name.fcov-index.html* and stores the other report files under *simv.vdb/reports/name.fcov*.

If you give a path name, the last component of the path is used as the base name. So the report files are stored under *path/name* and the index file is at *path/name.fcov-index.html*.

If this option is not included, the report files are stored under *simv.vdb/reports/report.fcov* and the index file is named *report.fcov-index.html*.

Customizing the Report with Tcl Commands

After you enter fcovReport, you can enter Tcl commands to modify the report. These commands also work in assertCovReport that you use for SystemVerilog assertions coverage reports. See [“Tcl Commands For SVA And OVA Functional Coverage Reports”](#) on page 23-49.

Using OVA with Third Party Simulators

Synopsys has developed OVA_{sim}, a PLI application that enables you to run non-Synopsys simulators using OVA (OVA). With OVA_{sim}, you can create a powerful system of assertion-based checkers. Because the checkers are compiled independently of a specific simulator, they can be used with any major simulator, packaged with IP designs, and shipped to any customer.

OVA_{sim} works by compiling the OVA code into a shared object and creating a wrapper file that forms the link between the checkers and the design. This wrapper file provides ports to the signals of interest in the design and contains all necessary PLI calls. Also, because the OVA code specifically refers to the ports of the wrapper file, it is largely insulated from design changes. The wrapper file and shared object generated by OVA_{sim} are compiled and run as a part of the design by the simulator.

For more information on OVA_{sim}, contact vcs_support@synopsys.com.

Inlining OVA in Verilog

Inlined OVA enables you to write any valid OVA code within a Verilog file using pragmas. In most usage cases, the context is inferred automatically and the OVA code will be bound to the current module.

You can use this process with or without regular OVA files. The results for both inlined and regular (OVA) assertions are reported together.

Inlined OVA is enabled in VCS by the `-ova_inline` command line switch.

Specifying Pragmas in Verilog

Inlined OVA is specified in Verilog code using pragmas. Several different forms are accepted, including C and C++ style comments, and modified C++ multi-line comments.

The general C++ style form is as follows:

```
/* ova first_part_of_pragma  
...  
last_part_of_pragma  
*/
```

You can also use the following modified C++ approach:

```
//ova_begin  
// pragma_statement  
//...  
//ova_end
```

For a single-line pragma, you can use the following C form:

```
// ova pragma_statement;
```

Assertions can be placed anywhere in a Verilog module and they use predefined units, including those in the Checker Library.

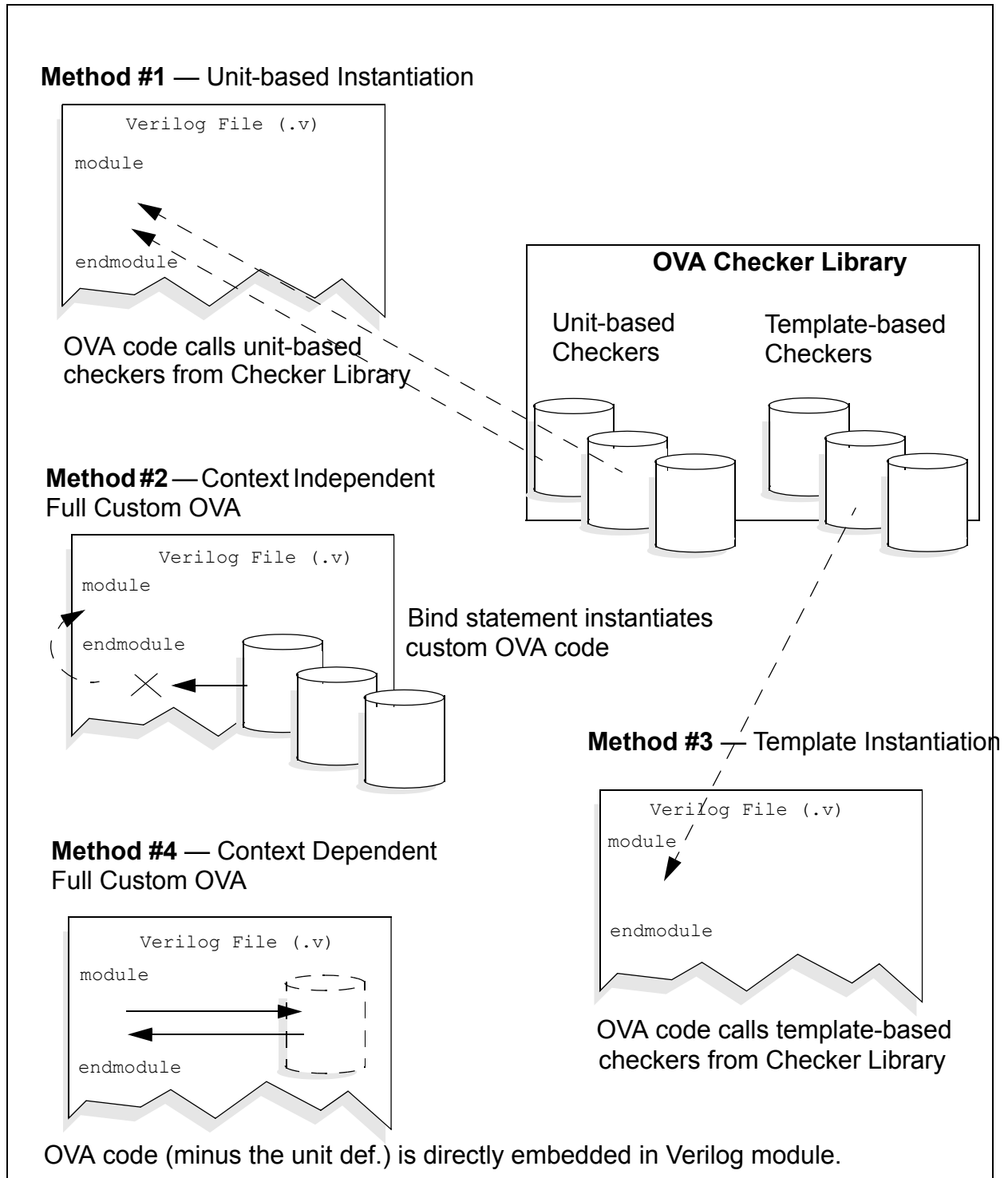
Methods for Inlining OVA

There are four basic methods you can use to inline OVA within Verilog:

- Unit Instantiation Using the Unit-based Checker Library (recommended for using Synopsys-developed OVA checkers)
- Context-Independent Full Custom OVA (uses custom-developed OVA code that resides in a Verilog file)
- Template Instantiation Using the Template-Based Checker Library
- Context-Dependent Full Custom OVA

These methods are described in detail throughout this section. Figure 20-2 provides an overview of these methods.

Figure 20-2 Methods for Inlining OVA within Verilog



Unit Instantiation Using the Unit-Based Checker Library

The easiest and most efficient method to inline OVA is to instantiate a unit-based checker from the OVA Checker Library (For more information on the Checker Library, see the *OpenVera Assertions Checker Library Reference Manual*). The context of the checker is automatically inferred based upon the location of the OVA pragma statement within a module of Verilog code. To use this method, you must include the bind keyword within a valid OVA pragma statement.

The syntax options for specifying unit-based assertions are as follows:

C++ Style:

```
/* ova
bind unit_name [inst_name] [#(param1, ..., paramN)] [(port1, ..., portN)];
*/
```

Modified C++ Style:

```
//ova_begin
//bind unit_name [inst_name] [#(param1, ..., paramN)] [(port1, ..., portN)];
//OVA_END

*/
```

C Style:

```
// ova bind unit_name [inst_name] [#(param1, ..., paramN)] [(port1, ..., portN)];
```

Note:

In all syntax styles, you can also split a pragma statement into separate lines as follows:

```
//ova bind
//ova unit_name [instance_name]
//ova [#(parameter1, ..., parameterN)]
//ova [(port1, ..., portN)];
```

The following example shows how to instantiate a checker, called `ova_one_hot`, from the OVA Checker Library:

```
module test();
reg [3:0] x;
wire clk;
wire a,b;
wire error;
// other verilog code
// ova bind ova_mutex (1'b1,clk,a,b);
/* ova bind
   ova_forbid_bool (error,clk);
*/
// ova_begin bind
//   ova_one_hot
//       #(0, // strict
//         4) // bit width
//       (1'b1, // enable
//        clk, // clock
//        x); // data
// ova_end
// other verilog code
endmodule // module test
```

Uses a single-line, C style pragma to instantiate the `ova_mutex` checker from the Checker Library, and checks for mutual exclusive of a and b.

Uses a multi-line C++ style pragma to instantiate `ova_forbid_bool`, and check that an error is never asserted.

Uses a multi-line modified C++ style pragma to instantiate `ova_one_hot` and checks that signal x has only 1 bit.

Instantiating Context-Independent Full Custom OVA

You can inline OVA within Verilog by instantiating independent custom OVA code located in the Verilog file but outside a Verilog module definition. The unit definition associated with the code must be specified outside a Verilog module.

The following example demonstrates this method:

```

module test();
reg [3:0] x;
wire clk;
wire a,b;
// ova bind (my_mutex(clk,{a,b}));
wire error;
// verilog code
endmodule // module test

```

Binding from inside a module

```

/* ova
unit error_check (logic clk, logic error);
clock posedge clk {
    event e1 : error == 1;
}
assert a1 : forbid(e1);
endunit
bind module test : error_check(clk,error) ;

unit my_mutex (logic clk, logic [1:0] x);
clock posedge clk {
    event e1 : x != 2'b11;
}
assert a1 : check(e1);
endunit
*/

```

Binding from outside a module

Two units are defined:
error_check and my_mutex

In the previous example, the bind statement (`// ova bind my_mutex(clk,{a,b});`) calls independent OVA code located outside Verilog module. You can instantiate the independent OVA code as many times as needed anywhere in the Verilog code. The context of the OVA code within the Verilog code is automatically inferred based upon the location of the bind statement.

Template Instantiation Using the Template-Based Checker Library

You can instantiate any template-based checker from the Checker Library in your Verilog code. The context of the checker is automatically inferred based on the location of the call from within Verilog.

Note the following construct usages and limitations in template Instantiation:

- Clocks must use edge expressions (unit-based checkers use Verilog style ports)
- You can specify the default clock in conjunction with the `check_bool` and `forbid_bool` checkers, however, it does not work with other templates or units. The following example shows a supported default clock:

```
//ova clock posedge clk;
```

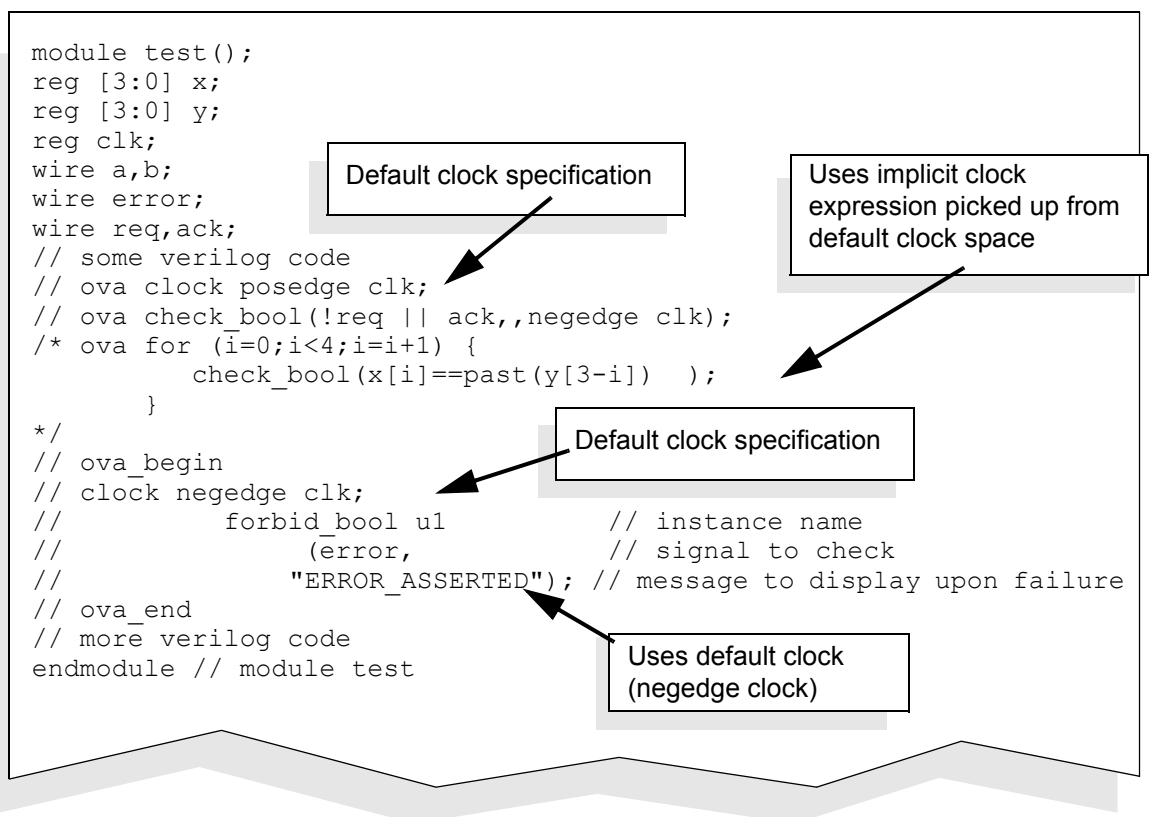
Note that each sequence or boolean expression is associated with a clock. The clock determines the sampling times for variable values.

- Both for loops and nested for loops are supported, as shown below:

```
for (name=expr;name op1 expr;
      name=name op2 expr)
{
for loop body
}
```

- You cannot mix a template instantiation and unit instantiation within the same OVA pragma using the multi-line C++ and modified C++ pragma specification formats. You can, however, specify a template instantiation and a unit instantiation using separate single-line C pragmas.

The following example demonstrates how to implement template instantiation using the template-based checker library:



The example calls the `check_bool` template from the OVA checker library. Note that the default clock, (`// ova clock posedge clk;`), must be a local signal, and can be boolean expression. It works only for the "check_bool" and "forbid_bool" templates, and does not work with other templates.

Inlining Context-Dependent Full Custom OVA

You can directly inline any custom OVA code, except for the unit definition, within a Verilog module. In this case, the unit and binding definitions are implicit.

The following example demonstrates this method:

```
module test();
reg clk;
wire a,b;
// other verilog code
// ova_begin
// clock posedge clk {
//   event e1 : a #1 b;
// }
// assert a1 : forbid(e1);
// ova_end
/* ova
   clock posedge clk {
       event e2 : ended(e1) #2 b;
   }
   assert a2 : forbid(e2);
*/
// more verilog code
endmodule // module test
```

Uses modified C++ style pragma to specify custom OVA code that defines event e1 and assertion a1.

Uses C++ style pragma to specify custom OVA code that defines event e2 and assertion a2.

Case Checking

You can perform two types of case checking when using inlined OVA:

- *Parallel* — The actual case selection value must select exactly one of the specified cases in the statement.
- *Full* — The actual case selection value must fall into the range specified by the case items. A case containing the default item is by definition full.

To control parallel case checking, use the `parallel_case` statement:

```
//ova parallel_case on | off ;
```

When `on` is specified, all case statements until the end of the module and the entire hierarchy underneath will be checked to ensure the rules of “parallel case” execution, unless overridden by another command or a local override pragma.

If the `off` argument is specified, parallel case checking is disabled unless overridden by another command or a pragma associated with a case statement.

When the pragma is not specified the default is *off*.

To control full case checking:

```
//ova full_case on | off ;
```

When `on` is specified, all case statements until the end of the module and the entire hierarchy underneath will be checked to ensure the rules of “full case” execution, unless overridden by another command or a local override pragma.

If the `off` argument is specified, full case checking is disabled unless overridden by another command or a pragma associated with a case statement.

When the pragma is not specified, the default is `off`.

The following rules govern parallel and full case commands:

- The commands must precede any module instance and any case statement in the module.
- If such a command is not provided, the default from the parent module is taken. (It can be by default *off*.) Also, you must make sure that every instance of the child module receives the *same* specifications. To avoid this limitation, widely used modules should include case-checking specifications.
- If a case statement appears in a function or a task, the module that contains the function or task declaration determines the default case checks (unless overridden by a local case pragma on the case statement).

Context-Dependent Assertion Pragmas

OVA includes three local assertions that depend on the context in which they are placed in the Verilog code:

```
// ova parallel_case;  
  
// ova full_case;  
  
// ova no_case;
```

The pragma must be placed immediately following a *case (expression)* statement. If placed anywhere else, an error will be reported. They apply only to the associated case statement. The defaults continue to apply elsewhere.

The `parallel_case` and `full_case` statements enable their own type of case check on the associated case statement and disable the other type *unless* both are present.

The `no_case` statement disables any case checking on that case statement.

The following rules govern assertion pragmas:

- These assertions verify the correct operation of case statements. They are triggered any time the case statement is executed. That is, no sampling of signals by a clock takes effect.
- The same pragma may be applied multiple times within a module. Each appearance will be considered an invocation of the assertion for its associated statement.
- If no such pragma assertion is associated with a case statement then the default setting established by an `// ova` command takes effect.
- Each `// ova` pragma may contain only one assertion terminated by “;”.
- Multiple case pragmas can be associated with a case statement, each on a separate line.
- The `no_case` pragma takes precedence over any other specification.

General Inlined OVA Coding Guidelines

Note the following guidelines when coding inlined OVA:

- Since OVA pragmas are declarative in nature, they do not need to be placed within procedural code, for example, within tasks, functions, and always/initial/forever blocks.
- Cross-module References (XMRs) are not allowed to concurrently reside within a custom OVA description.
- Template instantiation will be treated as an OVA description (the XMR restriction also applies).
- Unit binding is allowed inside 'module endmodule', but the keyword 'bind' needs to be specified in order to distinguish it from template instantiation. Unit binding (with the keyword 'bind') can have XMRs in its connections; however, using XMRs in connections without the bind keyword will cause a warning statement. Inlined bindings have the same requirements and limits as bindings created in an OVA source file.
- Inlined OVA cannot be read directly by third-party simulators. You can, however, use VCS to produce output files that can be read by third-party simulators.
- Each instance must be terminated by a ";" (semi-colon).
- Both positional and named (explicit) association of actual arguments are supported. However, the two styles cannot be used simultaneously in an instance.

Using Verilog Parameters in OVA Bind Statements

This section describes the enhancement to VCS that allows Verilog parameters to be used in temporal context in OVA bind statements. This enhancement is under an optional switch. At this time, use of Verilog parameters is limited to inline bind statements, but simple workarounds exist to allow binding of non-inline OVA while still allowing full use of Verilog parameters.

For purposes of this document, a value is used in *temporal context* if it is used as a delay (or delay range), a sequence length (or length range), or as a repeat count in an OVA event or assert statement. A value is used in *static context* if it is used to compute the bit width or bounds of a vector in an OVA unit statement.

Use Model

The current OVA use model allows the use of Verilog parameters only in static context. That is, the vector widths of an OVA unit may be defined using Verilog parameters. This continues to be the case even after this enhancement.

However, the OVA compiler does not have access to the Verilog parameters at the time the OVA state machine is compiled. The values of the parameters must be extracted and passed to the OVA compiler in order to use these values in temporal context.

The enhancement described in this document will cause the Verilog parameters to be expanded at the time the inlined OVA is processed by the OVA extractor, since the OVA binds and the design hierarchy are both available at that time. This implies that Verilog parameters can only be used in temporal context if the parameter was first expanded during the inlining phase. For that reason, only inline OVA binds will be able to pass Verilog parameter values to a unit if the parameter is to be used in a temporal context.

Enabling Verilog Parameter Expansion

To enable the substitution of Verilog parameters during inlining, the `-ova_exp_param` option must be used on the command line at compile time. In addition, the `-ova_inline` option must be enabled in order to enable the OVA inline processing.

Limitations on the Input

Only those binds found in single-line pragmas will be scanned for Verilog parameters (this is an internal limitation of the OVA inline parser). In order to use a Verilog parameter in temporal context via an OVA bind statement, the bind statements must be of the form:

```
// ova bind module foo : unitA u1 #( count ) ( ... );  
// ova bind module bar : unitB u2 #( delay ) ( ... );  
// ova bind ...etc...
```

Multiple-line inline pragmas (mostly used for full-language inlining) follow a different flow when they are processed and will not work if Verilog parameters are used in temporal constructs (but they will continue to work correctly for binds containing constants or Verilog parameters used only in static context).

For example, the following binds would not be legal if count and delay were used by the bound unit in temporal context:

```
/*
ova bind module foo : unitA u1 #( count ) ( ... );
ova bind module bar : unitB u2 #( delay ) ( ... );
*/
// ova_begin
ova bind module foo : unitA u1 #( count ) ( ... );
ova bind module bar : unitB u2 #( delay ) ( ... );
// ova_end
```

The parameter names are resolved in the context of the module to which the unit is bound. If the unit parameter is used in temporal context, the Verilog parameter bound to that unit must not use a cross-module reference (cross-module references are allowable if the parameter is only used in static context). Both module and instance binding is supported by this enhancement. But only binds directly found in pragmas in the Verilog file will be recognized. Bind statements inside files which are included in one or more Verilog files via ``include` will not be processed.

Table 20-3 Verilog Parameter Use Model Summary

Parameter context:	Static (bus width)	Temporal (delay, repeat, ...)
Type of bind	Module or instance	Module or instance
Location of bind	Inline or separate file	Inline flow only
XMR parameters allowed	Yes	No
Pragma format	Single or multiple line	Single line only (<code>// ova bind</code>)
Can put bind in include file	Yes	No

Recommended Methodology

To make use of Verilog parameters in temporal context, the binds that use such parameters must follow the inlined OVA flow. For OVA binds which are already inlined into the RTL, no additional work is required. It is not necessary, however, that the bind statements be inserted directly into the RTL source. Rather, a separate file (such as `dummy.v`) could be created to hold all the binds in the design that come from separate (non-inlined) OVA files. Giving this file a ".v" extension and passing it to the compiler with `-ova_inline` and `-ova_exp_param` enabled is enough to get the inline preprocessor to substitute the parameter value for the parameter name before passing the converted OVA code to the OVA compiler.

To avoid future problems, we also recommend moving the binds that are not already inlined into one or more of these dummy Verilog files. That way, if the contents of the OVA unit change at some point in the future (for example, a new parameter used in temporal context is added when there was previously no such parameter), the bind will continue to work as expected. Also, while other OVA code (such as units or templates) can be added to these dummy Verilog files, we do not recommend this, as there is some chance of confusing the inline processor (which, at this point, does not use a very sophisticated parser).

Caveats

The inline pre-processor writes the extracted inlined OVA into a file called `generated.ova` under the `ova.vdb` directory hierarchy. In the past, this file could be copied into a user-level file and used in subsequent simulation runs as a block of extracted OVA. This can still be done, to a certain extent. However, one of the following must be true:

- The `-ova_exp_param` option is not enabled.
- Any modules to which units using Verilog parameters are bound occur only once in the design
- Multiple instances of any modules to which units using Verilog parameters are bound use the same value for each parameter across all instances.

In other words, if any one module is instanced multiple times with different parameter values for two or more of the instances, then the parameter expansion that occurs at the time the inlined OVA is extracted will render the generated.ova file unusable for other purposes.

Post-processing Flow

A small change to the post-processing flow is necessary in order for this enhancement to have an effect on the OVA code compiled for post-processing. Recall that, normally, inlined OVA is not supported in the post-processing flow. However, it is still possible to invoke the inline pre-processing step as part of this flow.

Use Model

The existing post-processing flow consists of three basic steps:

1. Compilation of the design (Verilog) files using `-ova_PP`,
2. Compilation of the OVA source files using `-ova_RPP`
3. Replay of the saved simulation vectors with `OVAPP`

The first step generates a skeleton Verilog file containing the hierarchy of the design and the nets converted to registers (for playback). It also preserves the Verilog parameters. This skeleton Verilog file must be compiled in the second step before the playback can occur. It is this compilation step that we will exploit to allow Verilog parameters to be used in the post-processing flow.

The change to the flow involves the same dummy Verilog file that was discussed in the sections above. This file, which is disguised as a Verilog file but, in reality, contains nothing but inlined bind statements, is passed to the compiler during the `-ova_RPP` (second) step. In addition, the `-ova_inline` and `-ova_exp_param` options are added to the `-ova_RPP` compile step. This will cause the inline pre-processor to pick up the binds along with the remainder of the OVA code (which should still be in separate ".ova" files) and expand the parameters according to the values found in the skeleton design file.

OVA System Tasks and Functions

OVA system tasks and functions enable you to set conditions and control the monitoring of assertions, and specify the response to failed assertions.

Setting and Retrieving Category and Severity Attributes

You can use the following system tasks to set the category and severity attributes of assertions:

```
$ova_set_severity("assertion_full_hier_name",  
    severity)
```

Sets the severity level attributes of an assertion. The severity level is an unsigned integer from 0 to 255.

```
$ova_set_category("assertion_full_hier_name",  
    category)
```

Sets the category level attributes of an assertion. The category level is an unsigned integer from 0 to $2^{24} - 1$.

You can use the following system functions to retrieve the category and severity attributes of assertions:

```
$ova_get_severity("assertion_full_hier_name")  
    Returns unsigned integer.
```

```
$ova_get_category("assertion_full_hier_name")  
    Returns unsigned integer.
```

After specifying these system tasks and functions, you can start or stop the monitoring of assertions based upon their specified category or severity. For details on starting and stopping assertions based on their category specification, see [Category and Severity-Based Monitoring on page 15-40](#). For details on starting and stopping assertions based on their severity specification, see [“Starting and Stopping the Monitoring of Assertions” on page 20-70](#).

Starting and Stopping the Monitoring of Assertions

There are several methods you can use to start and stop the monitoring of assertions:

- **Global Monitoring** — Starts or stops assertions based upon their hierarchical location level at or below specified modules, entities, or scopes (note that these tasks overwrite any other tasks that control monitoring).
- **Category-Based Monitoring** — Starts or stops assertions based upon their category specifications.
- **Severity-Based Monitoring** — Starts or stops assertions based upon their severity specifications.
- **Name-Based Monitoring** — Starts or stops assertions based on the specified name of the assertion.

Global Monitoring

The `$ova_start` and `$ova_stop` system tasks enable you to control the monitoring of assertions based on the specified hierarchical location of the module, scope, or entity. Note that these tasks are specified on a global level, and overwrite any other tasks that start or stop assertion monitoring.

To start level-based monitoring, use the `$ova_start` system task:

```
$ova_start((levels [, module_or_scope_arguments]));
```

The integer argument *levels* indicates how many levels in the hierarchy at and below the specified modules, entities, and scopes the OVA monitoring is to start. Zero indicates all levels below the specified ones. Negative values are not allowed.

Within OVA and Verilog code, arguments can be specified for one or more module names as quoted text strings (e.g., "*module_name*") and/or instance scopes as Verilog scope names (unquoted), separated by commas.

Scope resolution follows the Verilog Standard (IEEE Std 1364-2001) Section 12.5. That is, if a scope argument is specified, then the path name is first considered as relative to the module where the task was called. If it does not match any valid relative path, it is matched against paths one level up. If there is no match then it is matched against paths one more level up, etc. For example, if scope *a.b.c* is an argument of a task called from module *m*, the scope name is first interpreted as a path starting at an instance *a* in *m*. If such a path does not exist, it is applied to a parent module of *a*. If the path does not exist in the root module of *m*, the path is an error.

If there are no arguments, i.e., `$ova_start`, the task applies to the entire design. It is equivalent to `$ova_start(0)`.

Note that the system task is ignored if monitoring was already started in the specified scope, module, or entity.

To stop monitoring, use the `$ova_stop` system task:

```
$ova_stop[(levels [, module_or_scope_arguments])];
```

Similar to the `$ova_start` system task, the integer argument *levels* indicates how many levels in the hierarchy at and below the specified modules, entities, and scopes the OVA monitoring is to stop. Zero indicates all levels below the specified ones.

Arguments can be specified for one or more modules, entities, and instance scopes as in the case of `$ova_start`. The effect of a module or entity argument is to stop monitoring assertions for all instances of the module and their respective hierarchies under each instance as indicated by levels. The effect of a scope argument is to stop monitoring assertions for that instance scope and the hierarchy under it as indicated by levels. If module, entity and scope arguments are omitted then levels applies to all the root modules of the design.

If there are no arguments, i.e., `$ova_stop`, the task applies to the entire design. It is equivalent to `$ova_stop(0)`.

The system task is ignored if the monitoring has already been stopped in the specified scope or module.

OVA monitoring is started automatically at the beginning of simulation. This is for compatibility with the non-inlined version where monitoring starts immediately at time 0. To control monitoring with the `$ova_start` and `$ova_stop` tasks, `$ova_stop` must be issued at time 0 — e.g., in an initial block.

This is similar to the use of `$dumpvars` where dumping starts immediately after calling this task and can be inhibited by calling `$dumpoff` right after. Here, the equivalent of calling `$dumpvars` is implicit in the start of simulation.

Examples:

```
$ova_start;    // Start assertions in the whole design.

$ova_start(0) // Start assertions in the whole design.

$ova_stop(3)  // Stop assertions in all top modules
              // to three levels below.
```

```
$ova_start(1, "mod1") // Start assertions in all
                    // instances of module mod1 at that level only.

$ova_stop(0, i1.mod1) // Stop assertions in all of
                    // the hierarchy at and below scope i1.mod1.

$ova_stop(3, i1.mod1) // Stop assertions in the hierarchy
                    // started at scope i1.mod1 to a depth
```

Category and Severity-Based Monitoring

To control category and severity-based assertion monitoring, you can use the following system tasks:

```
$ova_category_start(category)
    Starts all assertions associated with the specified category. The
    category level is an unsigned integer from 0 to  $2^{24} - 1$ 
```

```
$ova_category_stop(category)
    Stops all assertions associated with the specified category.
```

```
$ova_severity_start(severity)
    Starts all assertions associated with the specified severity level.
    The severity level is an unsigned integer from 0 to 255.
```

```
$ova_severity_stop(severity)
    Stops all assertions associated with the specified severity level.
```

Name-Based Monitoring

To control monitoring of assertions based upon the specified names of assertions, use the following system tasks:

```
$ova_assertion_stop("fullHierName")
    Stops the monitoring of the specified assertion (string).
```

```
$ova_assertion_start("fullHierName")
```

Starts the monitoring of the specified assetion (string).

Controlling the Response To an Assertion Failure

You can specify a response to an assertion failure, based upon its severity or category, using the `$ova_severity_action` or `$ova_category_action` system tasks. The syntax for these two tasks is as follows:

```
$ova_severity_action(severity, action);
```

or

```
$ova_category_action(category, action);
```

Note the following syntax elements:

severity

Severity to be associated with the action.

category

Category associated with the action.

action

Can be specified as "continue", "stop" or "finish". The action, which must be quoted as shown, is associated globally with the specified severity level, for all modules and instances. The default is "continue". The actions are specified as follows:

- "stop" — Stops the simulation with \$stop semantics. All OVA attempts are also suspended and can be resumed.
- "finish" — Terminates the simulation with \$finish semantics.

- "continue" — Outputs the message associated with the assertion failure but otherwise has no effect on the simulation.

Display Custom Message For an Assertion Failure

You can display a custom message upon failure of an `assert` statement `check` or `forbid`. The `assert` statement accepts an action block as follows:

```
assert [name] [[index]] : check | forbid
      (sequence_expr | event_name [, message[, severity [,
      category]]]) [action_block];
```

Where:

```
action_block ::= [else display_statement]
```

`display_statement` is similar to the standard `$display` system task:

```
$display(formatting_string, list_of args);
```

If both `message` and a `$display` are specified, then both are output upon assertion failure.

The following expression and system function calls are allowed in the `list_of args`.

expression

Output the value of the (bitvector) expression at the time of assertion failure (or success). It can be any expression over formal port/parameter names and OVA variables. It CANNOT contain any OVA - specific operators such as the temporal operators `past`, all edge operators, and `count`.

The evaluation of the uses the sampled values of the variables as in the assertion.

`$ova_current_time`

Returns current simulation time as a 64-bit entity. (Same as type "time" in Verilog.)

`$ova_start_time`

Returns assertion start time as a 64-bit entity. (Same as type "time" in Verilog.)

`$ova_global_time_unit`

Returns global time unit as a string. (The definition of a string is the same as in Verilog.)

Task Invocation From the CLI

Any CLI mode (command option `+cli+n` where $n = 1-4$ or `+cli` with no numerical value):

`$ova_stop levels`

Applied down to depth levels with respect to the present scope.

`$ova_start levels modname`

Applied to all instances of module "*modname*" down to a depth of levels. Note that no XMR is allowed. However module names are allowed without a quoted string.

`$ova_severity_action level "action"`

Sets the action for severity level where action is continue, stop, or finish.

The commands can also be issued using the format used in task calls that is, arguments in parentheses separated by commas. Module names must be quoted as in the task calls.

Note that you can specify all OVA system tasks and functions described in this chapter (`$ova_set_severity`, `$ova_set_category`, etc.) at the CLI using the documented syntax.

Debug Control Tasks

When dumping of OVA assertion results for viewing using DVE is enabled using the `-ova_debug` or `-ova_debug_vpd` options, the debug information is dumped for all enabled OVA assertions (enabled using `$ova_start` or by default from time 0 if not stopped by `$ova_stop`).

Dumping can be turned off by calling the following task in the Verilog code:

```
$ovadumpoff;
```

It can be turned on again by calling the following system task:

```
$ovadumpon;
```

If dumping of Verilog signals is not enabled by calling the system task `$dumpvars`, OVA dumping will also include all signals that are referred to by the assertions. This information is placed in a VPD file, `vcdplus.vpd`, in the simulation run directory.

Calls From Within Code

The `$ova_start`, `$ova_stop`, and `$ova_severity_action` system tasks are provided to control the monitoring of assertions, such as delaying the start of assertion monitoring and terminating the monitoring based on some conditions. For example, you can start monitoring after some initial condition, such as reset, has been satisfied.

To start monitoring, use the `$ova_start` system task:

```
$ova_start[(levels [, module, entity, or scope arguments]);
```

Whenever this statement is executed, assertion monitoring starts. The command is ignored if monitoring was already started in the specified scope, module, or entity.

The integer argument *levels* indicates how many levels in the hierarchy at and below the specified modules, entities, and scopes the OVA monitoring is to start. Zero indicates all levels below the specified ones. Negative values are not allowed.

Within OVA and Verilog code, arguments can be specified for one or more module names as quoted text strings (e.g., “`module_name`”) and/or instance scopes as Verilog scope names (unquoted), separated by commas.

Scope resolution follows the Verilog Standard (IEEE Std 1364-2001) Section 12.5. That is, if a scope argument is specified, then the path name is first considered as relative to the module where the task was called. If it does not match any valid relative path, it is matched against paths one level up. If no match then one more level up, etc. For example, if scope *a.b.c* is an argument of a task called from module *m*, the scope name is first interpreted as a path starting at an instance *a* in *m*. If such a path does not exist, it is applied to a parent module of *a*. If the path does not exist in the root module of *m*, the path is an error.

If there are no arguments, that is, `$ova_start`, the task applies to the entire design. It is equivalent to `$ova_start(0)`.

To stop monitoring, use the `$ova_stop` system task:

```
$ova_stop[(levels [, module, entity, or scope arguments])];
```

Whenever this statement is executed, the assertion monitoring stops. All attempts that are in progress are reset. The command is ignored if the monitoring has already been stopped in the specified scope or module.

The integer argument *levels* indicates how many levels in the hierarchy at and below the specified modules, entities, and scopes the OVA monitoring is to stop. Zero indicates all levels below the specified ones.

Arguments can be specified for one or more modules, entities, and instance scopes as in the case of `$ova_start`. The effect of a module or entity argument is to stop monitoring assertions for all instances of the module and their respective hierarchies under each instance as indicated by levels. The effect of a scope argument is to stop monitoring assertions for that instance scope and the hierarchy under it as indicated by levels. If module, entity and scope arguments are omitted then levels applies to all the root modules of the design.

If there are no arguments, that is, `$ova_stop`, the task applies to the entire design. It is equivalent to `$ova_stop(0)`.

OVA monitoring is started automatically at the beginning of simulation. This is for compatibility with the non-inlined version where monitoring starts immediately at time 0. To control monitoring with the `$ova_start` and `$ova_stop` tasks, `$ova_stop` must be issued at time 0, e.g., in an initial block.

This is similar to the use of `$dumpvars` where dumping starts immediately after calling this task and can be inhibited by calling `$dumpoff` right after. Here, the equivalent of calling `$dumpvars` is implicit in the start of simulation.

Examples:

```
$ova_start;           // Start assertions in the whole design.

$ova_start(0)        // Start assertions in the whole design.

$ova_stop(3)         // Stop assertions in all top modules
                    // to three levels below.

$ova_start(1, "mod1") // Start assertions in all
                    // instances of module mod1 at that level only.

$ova_stop(0, i1.mod1) // Stop assertions in all of
                    // the hierarchy at and below scope i1.mod1.
```

```
$ova_stop(3, i1.mod1) // Stop assertions in the hierarchy
                    // started at scope i1.mod1 to a depth 3.
```

To specify the response to an assertion failure, use the

`$ova_severity_action` system task:

```
$ova_severity_action(level, action);
```

Note the following syntax elements for `$ova_severity_action`:

level

Severity level to be associated with the action.

action

Can be specified as "continue", "stop" or "finish". The action, which must be quoted as shown, is associated globally with the specified severity level, for all modules and instances. The default is "continue". The actions are specified as follows:

- "stop" — Stops the simulation with `$stop` semantics. All OVA attempts are also suspended and can be resumed.
- "finish" — Terminates the simulation with `$finish` semantics.
- "continue" — Outputs the message associated with the assertion failure but otherwise has no effect on the simulation.

Developing a User Action Function

Instead of specifying "continue", "stop" or "finish" as the action argument to the `$ova_severity_action` system task, you can specify a function that you develop to perform the action. To enable this feature the new struct types:

Ova_AssertionSourceInfo

This struct has the following fields:

`lineNo`

Represents the line number in the file where the assertion was written.

`fileName`

Represents the filename where the assertion was written in HDL or OVA source code.

OvaAssertionData;

This struct has the following fields:

`Ova_AssertName`

This represents the full hierarchical name of the assertion.

`Ova_ExprType exprType`

This represents the type of assertion(event/check/forbid).

`Ova_AssertionSourceInfo srcBlock`

This represents the source file information for the assertion.

`unsigned int severity:8`

This represents the severity assigned to the assertion. It is an eight bit integer constant.

`category:24`

This represents the category assigned to the assertion. It is a 24-bit integer constant.

`Ova_String userMessage`

This represents the custom user message assigned to the assertion.

`Ova_TimeL startTime`

This represents the start time for the assertion as an unsigned long long.

`Ova_TimeL endTime`

This represents the end time for the assertion as an unsigned long long.

For non-temporal assertions, `startTime` and `endTime` will always be the same.

The following is the prototype of the user action functions:

```
typedef void (*OvaAssertionFailCB) (OvaAssertionData  
asData);
```

For example, a sample callback function would look like:

```
void my_user_action(OvaAssertionData assertionData)  
{  
:  
}
```


21

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 bring about a dramatic improvement in your productivity, especially when your designs become very large and complex.

This chapter covers the following topics:

- [Major Features Supported in Native Testbench OpenVera](#)
- [Getting Started With Native Testbench OpenVera](#)
- [Compiling and Running the OpenVera Testbench](#)

- [Testbench Functional Coverage](#)
- [Temporal Assertions](#)
- [Using Reference Verification Methodology with OpenVera](#)
- [Testbench Optimization](#)

Major Features Supported in Native Testbench OpenVera

The features supported in Native Testbench have mainly two origins: those that are related to the OpenVera language and others that are related to the technology itself. These features are listed in the following sections.

High-level Data Types

- Classes, with inheritance and polymorphism
- Fixed-size, dynamic (variable-size), and associative (flexible-size) arrays
- Strings with predefined string-manipulation methods
- Enumerated type, reg, integer, event
- Lists with predefined methods

Flow Control

- All sequential control constructs

- Concurrency, or spawning off of concurrent child threads from a parent thread, using fork-join with the return to the parent conditional on the completion of any, all, or none of the concurrent threads
- Concurrency control constructs:
 - `wait_child` to wait on the completion of all child threads
 - `wait_var` to block the execution of a thread until the specified variable changes value
 - `suspend_thread` to suspend a thread until other threads complete or block
 - Mailboxes to pass data between concurrent threads
 - Semaphores to prevent multiple threads from accessing any resource at the same time
 - Triggers to enable threads to trigger events on which other concurrent threads are waiting
 - Syncs to enable waiting threads to synchronize on triggers from triggering threads

Other Features

- Re-entrant tasks and functions with arguments passed by reference or by value; can also have arguments with default values
- Calls to HDL (design) tasks in OpenVera (testbench) code and to OpenVera (testbench) tasks in HDL (design) code
- Randomization with stability for generating random stimulus

- Constraint solver for use in generating constrained random stimulus
- Sequence generator for generating random stimulus streams
- Dynamic checks with clear, concise constructs called `expects` that have automatic error-message generation capability for writing self-checking testbenches
- Interfaces, containing interface signals, for providing the connectivity between signals in a testbench and ports or internal nodes of a design
- Virtual ports and binds for sharing of interface signals between functionally similar testbench signals, or grouping of interface signals into functionally similar bundles. Results in neat, efficient and easily reusable interfaces with substantially fewer interface signals.
- Separate compilation of testbench (LCA feature)
- Command-line interface (CLI) debugger (beta feature)
-

Getting Started With Native Testbench OpenVera

This chapter outlines the fundamental program structure used in all OpenVera programs as well as describes the details of a typical flow for Native Testbench. It also includes the compile and runtime options needed when you choose to compile the example testbench together with the design.

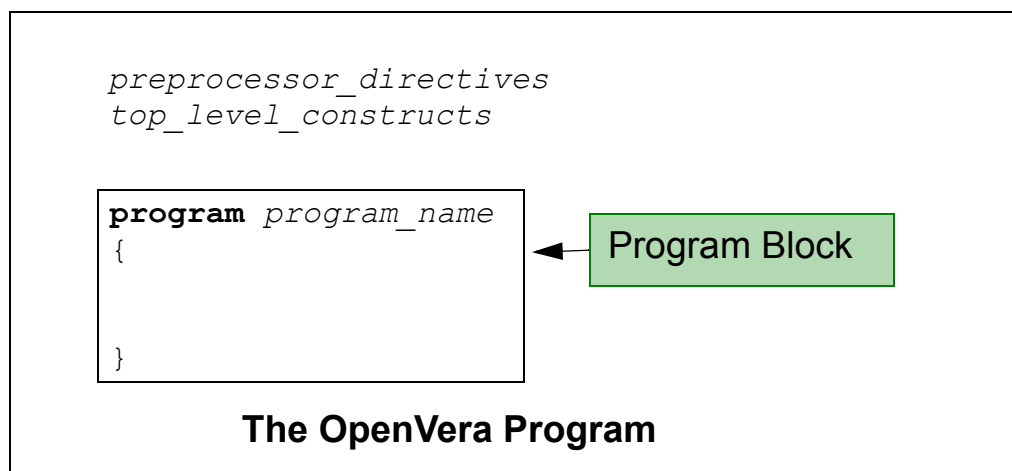
Basics of an OpenVera Testbench

This section outlines the fundamental program structure used in all OpenVera programs.

As Figure 21-1 illustrates, the constituents of an OpenVera program are:

- A required program block
- List of any preprocessor directives
- Top-level constructs

Figure 21-1 *Constituents of OpenVera Program*



Preprocessor Directives

An OpenVera source file can include another OpenVera source file by reference using the `include` construct. This construct can occur anywhere in the OpenVera program.

You must include the `vera_defines.vrh` file if you use predefined macros such as ON, OFF, HIGH, LOW.

```
#include <vera_defines.vrh>
```

Top Level Constructs

There can be any number of the following top level constructs:

- Enumerated type definitions
- Class definitions
- Out of block class method definitions
- Global task and function definitions
- Verilog task and function prototypes
- Interface declarations
- Virtual port and bind declarations

Program Block

The `program` keyword indicates the program block. This block contains:

```
program program_name
{
    variable declarations
    variable initializations
    program block code
}
```

The program block is where:

- Global variables are declared

- Testbench starts execution

Any variable defined in a task or function has local scope.

"Hello World!"

The following simple program prints "Hello World!" to standard output.

```
//hello.vr file

program hello_world
{
    string str;
    str = "Hello World!";

    printf("%s \n", str);    // printf() sends information
                            // to stdout
}
```

`//hello.vr file`, is a comment line. A single-line comment starts with the two slashes, `//`. This comment provides the name of the testbench file containing the program.

`program` is a keyword and indicates that this is where execution of the testbench begins.

`hello_world` is an identifier. Here, "hello_world" is the name of the program.

The left curly brace, `{`, must follow the program name. The right curly brace, `}`, indicates the end of the program.

`string str;` is the first statement in the program. `string` indicates the data type of the variable.

The variable, `str`, is a global string variable.

Statements are indicated with a semi-colon, ; .

The `str` variable is initialized on the next line (`str = "Hello World!";`).

A variable can be declared and initialized on the same line, that is, you can modify the example as follows:

```
string str = "Hello World!";
```

The `printf()` system task sends information to standard output during the simulation.

```
printf("%s \n", str);
```

`%s` is a string format specifier. `\n` indicates a line break. The `str` variable is passed to the task. The value of the `str` variable is printed out.

To see the output of this program, compile and run the `hello.vr` file as follows:

```
% vcs -ntb hello.vr
```

```
% simv
```

The output is, as expected:

```
Hello World!
```

The Template Generator

The typical flow for developing a testbench using the NTB template generator to start the process is as follows:

```
% ntb_template -t design_module_name [-c clock] filename.v \  
[-vcs vcs_compile-time_options]
```

Here:

design_module_name

The name of the module of your design.

The following template-generator command line illustrates all possible options:

```
% ntb_template -t arb -c clk arb.v -vcs vcs_option \  
[-vcs vcs_compile-time_options]
```

-t

Specifies the top-level design module name.

arb

Name of the top-level design module.

arb.v

Name of the design file.

-c

Specifies the clock input of the design.

-vcs

Specifies any VCS command that needs to be used in template generation. For example, if the top-level design module has ports listed in the Verilog-2001 format (v2k), then you need to specify the following: -vcs +v2k.

Multiple Program Support

Multiple program support enables multiple testbenches to run in parallel. This is useful when testbenches model standalone 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 standalone 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 at the VCS command line as a command line argument to `-ntb_opts`. For example:

```
vcs -ntb -ntb_opts config=configfile top.v
```

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.v, main2_dep2.vr ... main2_depN.vr contain definitions relevant to main2.vr, and so forth.

Use 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

Specifying All OpenVera Programs in Configuration File

You can specify all the OpenVera program files along with dependent OpenVera files in the configuration file using the configuration file syntax. The VCS command line for this method:

```
$vcs -ntb_opts config=configuration_filename
```

Specifying One OpenVera Program on Command Line

Specify one OpenVera program file with its dependent files on the command line. Place all other OpenVera program files, and their dependent files in the configuration file. The VCS command for this methods:

```
$vcs -ntb_opts \
  config=configuration_filename \
  testN.vr testN_dep1.vr testN_dep2.vr.
```

Note:

Specifying multiple OpenVera files containing the program construct at the VCS command prompt is an error.

Note:

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.

Compiling Multiple Programs

There are numerous scenarios for compiling multiple programs.

Program Files are Listed in the Configuration File

The VCS command line, for when there are two or more program files listed in the configuration file, is:

```
$vcs -ntb -ntb_opts config=configuration_filename
```

The configuration file, could be:

```
program main1.vr -ntb_define ONE
program main2.vr -ntb_incdir /usr/vera/include
```

One Program File is on the Command Line

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 -ntb_incdir \
    /usr/vera/include
```

The configuration file, in this example, is:

```
program main1.vr
```

In the above example, `main1.vr` is specified in the configuration file and `main2.vr` is specified on the command line along with the files need by `main2.vr`.

Compiling when there is a top-level module. If there is a top-level module having an instance of an OpenVera program, then you can compile in either of the following ways:

```
% vcs -ntb -ntb_opts config=configfile top.v
```

Note that all the program files are in the configuration file:

```
program test1.vr
program test2.vr
program test3.vr
```

or:

```
% vcs -ntb -ntb_opts config=configfile top.v test3.vr
```

The configuration file, in this example, is:


```
program test1.vr
program test2.vr
```

You can specify, at most, one OpenVera program file, along with its dependent OpenVera files, at the command prompt. Specifying more than one OpenVera file that contains a program construct at the command prompt is an error.

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`

-ntb_define macro

To run different OpenVera programs with different macro definitions, specify the macros name using the "-ntb_define" option. Multiple macro names are specified using the delimiter the "+".

```
ntb_options -ntb_define macro1
```

or

```
ntb_options -ntb_define macro1+macro2
```

-ntb_incdir directory

Specifies the path to the directory where *.vrh files to be included reside. Multiple include directories can be specified using the delimiter +.

Example:

```
ntb_options -ntb_incdir /usr/muddappa/include
```

or

```
ntb_options -ntb_incdir /usr/muddappa/include+/usr/vera/include
```

-ntb_opts no_file_by_file_pp

File by file preprocessing is done on each input file, feeding the concatenated results to the parser. To disable this behavior, use `no_file_by_file_pp`.

-ntb_opts tb_timescale=*value*

Specifies an overriding timescale for an OpenVera program. This option allows you to generate different timescale values for each OpenVera program.

Example:

```
-ntb_opts tb_timescale=1ns/10ps
```

-ntb_opts dep_check

Enables dependency analysis. Files with circular dependencies are detected. VCS issues an error message when it cannot determine which file to compile first.

-ntb_opts print_deps

Supplied with `dep_check`. Tells the compiler to output the dependencies for the source files to standard output or to a user specified file.

-ntb_opts use_sigprop

Compiling the OpenVera program with this option enables the signal property access functions.

The functions supported are:

- function integer `vera_is_bound()`

- function string vera_get_name()
- function string vera_get_ifc_name()
- function string vera_get_clk_name()
- function integer vera_get_dir()
- function integer vera_get_width()
- function integer vera_get_in_type()
- function integer vera_get_in_skew()
- function integer vera_get_in_depth()
- function integer vera_get_out_type()
- function integer vera_get_out_skew()

These functions are documented in the “Retrieving Signal Properties” section of the OpenVera LRM: Native Testbench manual.

-ntb_opts vera_portname

When specifying the vera_portname option to -ntb_opts, the naming convention is as follows:

1. The Vera shell module name is named vera_shell,
2. The interface ports are named ifc_signal,
3. The signals are named, for example, as: \ifc.signal[3:0]

The default behavior is:

The Vera shell module name is based on the name of the OpenVera program. Bind signals are named, for example, as: \ifc.signal[3:0]
The interface ports are named \ifc.signal

Summary

There are three major error scenarios, when specifying OpenVera files on the command line in addition to the configuration file:

1. An error is issued when the VCS command specifies multiple OpenVera files containing the program construct in addition to a configuration file.
2. An error is issued if the file on the command line which contains program, does not also have all of the files it is dependent on specified in the command line. If one file contains program whereas the other files contain tasks and functions required by the file containing program, the program is treated as part of multiple programs in the configuration file. That is, it is like an addition of one more programs in the configuration file.
3. An error is issued if none of the files specified on the command line contains the program construct.

Example Configuration File

```
program
    prog1.vr
    task11.vr
    task12.vr
    -ntb_opts tb_timescale=1ns/10ps
    -ntb_define NTB1
    -ntb_incdir /usr/muddappa/include

program
    prog2.vr
    task21.vr
    task22.vr
    -ntb_define NTB2+NTB3 -ntb_opts dep_check

program
    prog3.vr
    -ntb_incdir /usr/muddappa/include+/usr/vera/include
```

In this example configuration file, the prog1.vr, prog2.vr and prog3.vr files contain the OpenVera program construct. The OpenVera files task11.vr and task12.vr are two files needed by the program in the prog1.vr file. task21.vr and taks22.vr are OpenVera files needed by program in prog2.vr

Multiple program example:

```
top.v file
module duv_test_top;
    parameter simulation_cycle = 100;

    reg  SystemClock;

    wire [7:0]    d1;
    wire [7:0]    d2;
    wire          rst_;
    wire          clk;
    wire [7:0]    q1;
    wire [7:0]    q2;
    assign clk = SystemClock;

    duv_test1 u1(
        .SystemClock (SystemClock),
        .\duv1.d      (d1),
        .\duv1.q      (q1),
        .\duv1.rst_   (rst_),
        .\duv1.clk    (clk)
    );

    duv_test2 u2(
        .SystemClock (SystemClock),
        .\duv2.d      (d2),
        .\duv2.q      (q2),
        .\duv2.rst_   (rst_),
        .\duv2.clk    (clk)
    );

    duv dut(
        .d1 (d1),
```

```

        .d2 (d2),
        .rst_      (rst_),
        .clk       (clk),
        .q1 (q1),
        .q2 (q2)
    );

    initial begin
        SystemClock = 0;
        forever begin
            #(simulation_cycle/2)
            SystemClock = ~SystemClock;
        end
    end

endmodule

// duv.if1.vri
#ifdef INC_DUV1_IF_VRH
#define INC_DUV1_IF_VRH

    interface duv1 {
        input  [7:0]    q NSAMPLE ;
        inout  [7:0]    d PDRIVE #1 PSAMPLE #-1 ;
        output          rst_ PDRIVE ;
        input  clk CLOCK ;
    } // end of interface duv

    // hdl_node CLOCK "duv_test_top.clk";

#endif

// test1.vr
#include <vera_defines.vrh>
#include "duv.if1.vri"

program duv_test1 {
    printf("start of sim duv_test1\n") ;
    @1 duv1.rst_ = 0 ;
    @1 duv1.rst_ = 1 ;
    @1 duv1.rst_ = void ;

    @1 duv1.d = 1 ;

```

```

    @1 duv1.d = 2 ;

    @1 duv1.d = 3 ;
    @20 duv1.q == 8'h3 ;

    printf("end   of sim duv_test1\n") ;
}

// duv.if2.vri
#ifndef INC_DUV2_IF_VRH
#define INC_DUV2_IF_VRH
    interface duv2 {
        input    [7:0]    q PSAMPLE #-1;
        inout   [7:0]    d PDRIVE #1 PSAMPLE #-1 ;
        output          rst_ PDRIVE #1 ;
        input  clk CLOCK ;
    } // end of interface duv
#endif

// test2.vr
#include <vera_defines.vrh>
#include "duv.if2.vri"
program duv_test2 {
    printf("start of sim duv_test2\n") ;
    @1 duv2.rst_ = 0 ;
    @1 duv2.rst_ = 1 ;
    @1 duv2.rst_ = void ;
    @1 duv2.d = 1 ;
    @1 duv2.d = 2 ;
    @1 duv2.d = 3 ;
    @2 duv2.q == 8'h3 ;
    @1 duv2.d = 3 ;
    printf("end   of sim duv_test2\n") ;
}

// dut.v
module dut ( q1, d1, q2, d2, rst_, clk) ;
    input  [7:0] d1 ;
    output [7:0] q1 ;
    input  [7:0] d2 ;

```



```

    output [7:0] q2 ;
    input      rst_ ;
    input      clk  ;

    dff u1 (.q(q1), .d(d1), .rst_(rst_), .clk(clk)) ;
    dff u2 (.q(q2), .d(d2), .rst_(rst_), .clk(clk)) ;
endmodule
module dff (q, d, rst_, clk) ;
    input  [7:0] d ;
    output [7:0] q ;
    input      rst_ ;
    input      clk  ;
    reg  [7:0] q ;
    always @(posedge clk)
        q <= (!rst_)? 8'h00 : d ;
endmodule

```

```

// vlog.f
duv.v
top.v

// config.vr1
program
    ./test1.vr
program
    ./test2.vr

// 1_comp_run_ntb
#!/bin/csh -fX
vcs -ntb -f vlog.f -ntb_opts config=config.vr1
simv

```

Compiling and Running the OpenVera Testbench

This section describes how to compile your testbench with the design and how to compile the testbench independently from the design. It also describes compile-time and runtime options.

In order to ease transitioning of legacy code from Vera's make-based single-file compilation scheme to VCS, 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. The following sections describe how to do this.

Compiling the Testbench with the OpenVera Design

The VCS command line for compiling both your testbench and design is the following:

```
% vcs -ntb design_module_name.v module_name.test_top.v  
testbench_file.vr [vcs_compile-time_options]  
[ntb_compile-time_options]
```

The compilation results in a single executable simv that contains both testbench and design information.

For a list of NTB compile-time options, see Options for OpenVera Native TestBench in Appendix B.

The command line for running the simulation is as follows:

```
% simv +vcs_runtime_options
```

Compiling the Testbench Separate From the OpenVera Design

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 Verilog design 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.

Separate Compilation of Testbench Files for VCS

To compile the testbench file (tb.vr) and create the testbench shell (.vshell) and shared object (libtb.so) files, use the following syntax:

```
vcs -ntb_cmp [-ntb_options] [-ntb_noshell | \  
-ntb_shell_only] tb.vr
```

Here:

`-ntb_cmp`

Compiles and generates the testbench shell (*file.vshell*) and shared object files

Compile-time Options

`-ntb_sfname filename`

Specifies the file name of the testbench shell.

`-ntb_sname module_name`

Specifies the name of the testbench shell module.

`-ntb_spath`

Specifies the directory where the testbench shell and shared object files will be generated. The default is the compilation directory.

`-ntb_noshell`

Specifies not generating the shell file. Use this only when you are recompiling a testbench.

`-ntb_shell_only`

Generates only the `.vshell` file. Use this only when you are compiling a testbench separately from the design file.

Note:

When compiling the testbench files separately from the design, the following rules apply:

- Specify the ntb options should be specified only during testbench compilation. The exception is the `-ntb_vl` option that specifies that the DUT is compiled separately and which you should therefore specify while compiling the DUT.
- Specify 'C' files or other library files should be specified only while compiling the DUT. These options are ignored for testbench compilation.
- Specify all other tool options for both testbench and DUT compilation. For example, `-debug_all`, `-debug`, etc.

Compiling the Design, the Testbench Shell And the Top-level Verilog Module

Next you generate a `simv` for your design. The syntax for compiling the Verilog design file, `dut.v`, with the testbench shell (`tb.vshell`) and the top-level Verilog module (`test_top.v`) is the following:

```
vcs -ntb_vl dut.v name.test_top.v tb.vshell
```

Here:

```
-ntb_vl
```

Specifies the compilation of all Verilog files, including the design, the testbench shell file and the top-level Verilog module.

Example

```
% vcs -ntb_vl sram.v sram.test_top.v sram.vshell
```

Note:

Remember, if, for example, you used `-debug_all` when compiling the `.vr` file, you must include `-debug_all` on the `vcs` command line as well. For example:

```
% vcs -debug_all -ntb_vl dut.v name.test_top.v tb.vshell
```

Loading the Compiled Testbench On simv

Finally, load the compiled testbench shared object file, `libtb.so`, on `simv` using the following syntax:

```
% simv +ntb_load=path_name_to_libtb.so
```

Or

```
% simv +ntb_load=./libtb.so
```

Here:

```
+ntb_load
```

Specifies the testbench shared object file `libtb.so` to be loaded.

Example

```
% simv +ntb_load=test1/libtb.so
```

Limitations

- The `hdl_task inst_path` specification should begin with a top-level module (that is, not the DUT module name).
- When the `hdl_task` declaration has a `var` parameter, the corresponding task port in the DUT side must be “inout” (that is, it cannot be “output”).
- The port direction and size in the `hdl_task` specification must match the corresponding task in the DUT.
- You cannot rely on port-coercion to happen for `hdl_tasks` to correct the direction based on usage. For example, if one of the `hdl_tasks` port directions is specified as an input but is actually used as output, then it may not be able to be coerced into behaving as an output in the separate compile mode.
-

Compile-time Options

The following options can be used on the VCS command line for Native Testbench, whether compiling the testbench with the design, or separately from the design:

```
-ntb
```

Invokes Native Testbench.

`-ntb_cmp`

Compiles and generates the testbench shell (file.vshell) and shared object files. Use this when compiling the .vr file separately from the design file.

`+error+n`

Sets the maximum number (*n*) of errors before compilation failure.

`-f filename`

Specifies the file containing a list of all the source files to be compiled. Synopsys recommends that you append the .list extension to the filename to indicate that it is an NTB source list file. The `-f` option works equally well with both relative and absolute path names.

For example:

```
-----  
#file.list  
  
topTest.vr  
packet.vr  
checker.vr  
-----
```

```
% vcs -ntb design_module_name.v module_name.test_top.v \  
testbench_file.vr -f file.list
```

`-ntb_define macro`

Specifies any OpenVera macro name on the command line. You can specify multiple macro names using the + delimiter.

Examples:

```
-ntb_define macro1  
-ntb_define macro1+macro2
```


`-ntb_filext extensions`

Specifies an OpenVera file extension. You can pass multiple file extensions at the same time using the + delimiter.

Examples:

```
-ntb_filext .vr
-ntb_filext .vr+.vri+.vrl
```

`-ntb_incdir directory`

Specifies the include directory path for OpenVera files. You can specify multiple include directories using the + delimiter.

Examples:

```
-ntb_incdir ../src1
-ntb_incdir ../src1+../../src2
```

`-ntb_noshell`

Specifies not generating the shell file. Use this only when recompiling a testbench.

`-ntb_spath`

Specifies the directory where the shared object files will be generated. The default is the compilation directory.

`-ntb_vipext .extensions`

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`, `.vrhp` are not overridden, and are always in effect. You can pass multiple file extensions at the same time using the + delimiter. See [“Dependency-based Ordering in the Presence of Encryption” on page 21-43](#).

-ntb_vl

Specifies the compilation of all Verilog files, including the design, the testbench shell file and the top-level Verilog module.

-ntb_opts

Invokes a set of keyword arguments

Syntax:

```
-ntb_opts keyword_argument[+keyword_argument(s)]
```

Examples:

```
-ntb_opts check  
-ntb_opts no_file_by_file_pp+check
```

A list of keyword arguments is as follows:

check

Reports error, at compile time or runtime, for out-of-bound or illegal array access. For example:

```
// mda.vr  
program p  
{  
    bit ia[4][5];  
    integer i, j;  
  
    i = 4;  
    j = 5;  
    j = ia[i][j];  
}
```

This yields the following runtime error:

```
Error: Out of bound multi-dimensional array access  
(index number: 0, index value: 4) at time 0 in file  
mda.vr line 8
```

In cases where the out-of-bound index is directly specified as a constant expression as in the following example there is an error at compile time itself.

```
program p
{
    bit ia[4];
    integer i;
    i = ia[4]; // Or i = ia[1'bx];
}
```

The error message in this case is:

```
Error-[IRIMW] Illegal range in memory word
Illegal range in memory word shown below
"mda_2.vr", 7: ia[4]
```

Enables the built-in checker to identify any null-pointer access. For example, a list or class object being used without initialization.

Enables printing a thread stack trace upon a verification or null check error. This is very useful debug feature that helps to identify the caller of the thread in which the error occurs.

Prints the time and expected outputs of an expect statement upon a verification error.

`dep_check`

Enables dependency analysis. 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, file by file preprocessing is done on each input file, and the concatenated result is fed to the parser. To disable this behavior, use `no_file_by_file_pp`.

`print_deps`

The `-print_deps` option is supplied with `dep_check` telling the VCS compiler to output the dependencies for the source files to standard output or to a user specified file.

`-rvm`

Use `-rvm` when RVM is used in the testbench.

`tb_timescale`

Specifies an overriding timescale for the testbench. For example:

```
vcs -ntb_opts tb_timescale=1ns/1ps file.vr
```

This allows independent control of the testbench timescale as opposed to picking up the `timescale of the last `.v` file compiled.

Consider a case where you pass a timescale, `T1`, to VCS during the testbench compilation phase with the `-ntb_opts tb_timescale` option. If this timescale is different from the elaborated timescale, `T2`, used for the design, then in addition to `T1` you also have to pass `T2` during testbench compilation. You pass `T2` using the `-timescale` option. That is, the timescale passed to VCS during the testbench compilation phase should match the final elaborated timescale of the design.

`use_sigprop`

Compiling the testbench with the `+ntb_opts` keyword option, `use_sigprop`, enables the signal property access functions. For example, `vera_get_ifc_name()`.

Example:

```
vcs -ntb -ntb_opts use_sigprop test_io.v \  
test_io.test_top.v test_io.vr
```

The functions supported are:

```
function integer vera_is_bound(signal)
```

```
function string vera_get_name(signal)
```

```
function string vera_get_ifc_name(signal)
```

```
function string vera_get_clk_name(signal);
```

```
function integer vera_get_dir(signal);
```

```
function integer vera_get_width();
```

```
function integer vera_get_in_type();
```

```
function integer vera_get_in_skew();
```

```
function integer vera_get_in_depth();
```

```
function integer vera_get_out_type();
```

```
function integer vera_get_out_skew();
```

See Chapter 6 in the *OpenVera LRM: Native Testbench* for a full description of these functions.

```
vera_portname
```

When you pass the `vera_portname` option to `-ntb_opts` (that is, `-ntb_opts vera_portname`):

1. The Vera shell module name is named `vera_shell`.
2. The interface ports are named `ifc_signal`
3. Bind signals are named, for example, as: `\ifc_signal[3:0]`

Without this option, the default behavior is:

1. The Vera shell module name is based on the name of the OpenVera program.
2. Bind signals are named, for example, as: `\ifc.signal[3:0]`
3. The interface ports are named `\ifc.signal`

`covg_compat`

The `covg_compat` argument enables VCS to compile the source file in the old (pre-VCS 2006.06) semantics for coverage and disables all the new VCS 2006.06 coverage features such as:

- SV style auto binning
- New semantics for cross bins
- Accurate cross coverage computation
- Accurate hole analysis for cover points and crosses
- Default coverage goal changed to 100 percent

Example:

```
vcs -ntb_opts covg_compat [-other_compile_options]  
file.vr
```

The database generated using the `covg_compat` argument cannot be merged with a database generated without using it and vice versa.

Runtime Options

`-cg_coverage_control`

The `coverage_control()` system task (see the *OpenVera LRM: Native Testbench* for description of this task), coupled with the `-cg_coverage_control` runtime argument, provides a single-point mechanism for enabling/disabling the coverage collection for all coverage groups or a particular coverage group.

Syntax:

```
-cg_coverage_control=value
```

The values for `-cg_coverage_control` are 0 and 1. A value of “0” disables coverage collection, and a value of “1” enables coverage collection.

For an example, see [“Controlling Coverage Collection Globally” on page 21-51](#).

`+ntb_cache_dir`

Specifies the directory location of the cache that VCS-NTB maintains as an internal disk cache for randomization:

```
% simv +ntb_cache_dir=/u/myself/tmp_cache...
```

This cache improves performance and reduces memory use by reusing randomization problem data from previous runs. The default cache is called “`.__solver_cache__`” in the current working directory.

You can remove the contents of the cache at any time, except when VCS is running.

`+ntb_debug_on_error`

Stops the simulation when it encounters a simulation error. Simulation errors involve either an `expect` error or VCS calling the `error()` system task.

`+ntb_enable_solver_trace=value`

Enables a debug mode that displays diagnostics when VCS executes a `randomize()` method call. Allowed values are:

Value	Description
0	Disables tracing
1	Enables tracing
2	Enables more verbose message in trace

If `+ntb_enable_solver_trace` is specified without an argument, the default value is 1. If it is not specified, the default is 2.

`+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.

For example, if the constraints are:

```
x == 0;  
x > y;  
y < z+w;  
x == 5;  
y != 6;
```

If you specify `+ntb_enable_solver_trace_on_failure=2` (that is with value “2”), then you will get a report of just the inconsistent constraints.

For example:

```
x == 0;  
x == 5;
```

Allowed values are 0, 1, and 2. The default value is 2.

Value	Description
0	Disables tracing
1	Enables tracing
2	Enables more verbose message in trace

If `+ntb_enable_solver_trace_on_failure` is specified without an argument, the default value is 1. The default is 2, if the option is not specified.

```
+ntb_exit_on_error[=value]
```

Causes VCS to exit when value is >0. The value can be:

- 0: continue
- 1: exit on first error (default value)
- N: exit on nth error.

When value = 0, the simulation runs to completion regardless of the number of errors.

```
+ntb_load=path_name_to_libtb.so
```

Tells VCS which testbench shared object file libtb.so to load.

```
+ntb_random_seed=value
```

Sets the seed value used by the top level random number generator at the start of simulation. The `random(seed)` system function call overrides this setting.

Value can be any integer number.

`+ntb_solver_mode=value`

Allows you to choose between one of two constraint solver modes. When set to 1, the solver spends more pre-processing time in analyzing the constraints, during the first call to `randomize()` on each class. Subsequent calls to `randomize()` on that class are very fast. When set to 2, the solver does minimal pre-processing, and analyzes the constraint in each call to `randomize()`. Default is 2.

`+ntb_stop_on_error`

When the simulation encounters a simulation error, this option causes the simulation to stop immediately, and turns it into a CLI debugging environment. In addition to normal verification errors, `+ntb_stop_on_error` halts the simulation in case of runtime errors.

The default setting is to execute the remaining code within the present simulation time.

`+vera_enable_checker_trace`

Enables a debug mode that displays diagnostics when the `randomize(VERA_CHECK_MODE)` method is called.

`+vera_enable_checker_trace_on_failure`

Enables a mode that prints trace information only when the `randomize(VERA_CHECK_MODE)` returns 0.

`+vera_enable_solver_trace`

Enables a debug mode that displays diagnostics when the `randomize()` method is called.

+vera_enable_solver_trace_on_failure

Enables a mode that prints trace information only when the solver fails to compute a solution, usually due to inconsistent constraints. Legal values are 0, 1 and 2. When the value of the option is 2, the analysis narrows down to the smallest set of inconsistent constraints, thus aiding the debugging process.

+vera_solver_mode

Selects the Vera constraint solver to use. When set to 1, the solver spends more pre-processing time in analyzing the constraints, during the first call to `randomize()` on each class. Subsequent calls to `randomize()` on that class are very fast. When set to 2, the solver does minimal pre-processing, and analyzes the constraint in each call to `randomize()`.

Class Dependency Based OpenVera 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 `-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`, 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 {...}
```

Here classes are derived from base classes that are in the other file, respectively, or more generally, when the inheritance relationships project on to 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 the Presence of Encryption

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 (that shouldn't exist in unencrypted code to begin with). VCS splits Vera input files into those that are encrypted or declared as such by having extensions

.vrp, .vrhp, or as specified using option `-ntb_vipext`, 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 option `-ntb_vipext`. This implies that the extensions are considered OpenVera extensions similar to using `-ntb_filext` for unencrypted files. This causes those files and everything they include to be preprocessed in encrypted mode.

Testbench Functional Coverage

As chip designs grow more complex and testing environments become increasingly sophisticated, the emphasis is on testing the chip completely. With hundreds of possible states in a system and thousands of possible transitions, the completeness of tests must be a primary focus of any verification tool.

Traditional coverage models use a code coverage methodology. They check that specific lines of code are executed at some point in the simulation. However, this method has inherent flaws. For instance, you can be certain that a device entered states 1, 2 and 3, but you cannot be certain that the device transitioned from state 1 to 2 to 3 in sequence. Even such a simple example displays the limitations of code coverage methodology. With a sophisticated chip, such an approach is not adequate to ensure the integrity of the design.

VCS supports a functional coverage system. This system is able to monitor all states and state transitions, as well as changes to variables and expressions. By setting up a number of monitor bins that correspond to states, transitions, and expression changes, VCS is able to track activity in the simulation.

Each time a user-specified activity occurs, a counter associated with the bin is incremented. By establishing a bin for each state, state transition, and variable change that you want to monitor, you can check the bin counter after the simulation to see how many activities occurred. It is, therefore, simple to check the degree of completeness of the testbench and simulation.

VCS further expands this basic functionality to include two analysis mechanisms: open-loop analysis and closed-loop analysis.

- Open-loop analysis monitors the bins during the simulation and writes a report at the end summarizing the results.
- Closed-loop analysis monitors the bins during the simulation and checks for areas of sparse coverage. This information is then used to drive subsequent test generation to ensure satisfactory coverage levels.

Coverage Models Using Coverage Groups

The `coverage_group` construct encapsulates the specification of a coverage model or monitor. Each `coverage_group` specification has the following components:

- A set of coverage points. Each coverage point is a variable or a DUT signal to be sampled. A coverage point can also be an expression composed of variables and signals. The variables may be global, class members, or arguments passed to an instance of the `coverage_group`.
- A sampling event (a general event not just an OpenVera sync event) that is used for synchronizing the sampling of all coverage points.

- Optionally, a set of state and/or transition bins that define a named equivalence class for a set of values or transitions associated with each coverage point.
- Optionally, cross products of subsets of the sampled coverage points (cross coverage).

The `coverage_group` construct is similar to an OpenVera class in that the definition is written once and is instantiated one or more times. The construct can be defined as a top-level (file scope) construct (referred to as standalone), or may be contained inside a class. Once defined, standalone coverage is instantiated with the `new()` system call while embedded (contained) coverage groups are automatically instantiated with the containing object.

The basic syntax for defining a `coverage_group` is:

```
coverage_group definition_name [(argument_list)]
{
    sample_event_definition;
    [sample_definitions;]
    [cross_definitions;]
    [attribute_definitions;]
}
```

The syntax for defining an embedded `coverage_group` is:

```
class class_name
{
    // class properties
    // coverage
    coverage_group definition_name .... (same as external).
    // constraints
    // methods
}
```

For definitions, see the Coverage Group section in the *OpenVera LRM: Native Testbench* book.

Example 21-1 shows a standalone coverage_group definition and its instantiation.

Example 21-1 Defining and Instantiating a Standalone Coverage Group

```
interface ifc
{
    input clk CLOCK;
    input sig1 PSAMPLE #-1;
}

coverage_group CovGroup
{
    sample_event = @ (posedge CLOCK);
    sample var1, ifc.sig1;
    sample s_exp(var1 + var2);
}

program covTest
{
    integer var1, var2;
    CovGroup cg = new();
}
```

In this example, coverage_group CovGroup defines the coverage model for global variable var1, the signal ifc.sig1, and the expression composed of global variables var1 and var2. The name of the sampled expression is s_exp. The two variables and the expression are sampled upon every positive edge of the system clock. The coverage_group is instantiated using the new() system call.

For details on the syntax of both standalone and embedded coverage_groups and how to instantiate them, see the *OpenVera LRM: Native Testbench* book.

Measuring Coverage

VCS computes a coverage number (or percentage) for the testbench run as a whole. Here, the coverage number is referred to as “coverage”. The coverage for the testbench is the weighted average of the coverages of every coverage_group in the testbench. When per-instance data is available, VCS also computes an instance coverage for the testbench. That number is the weighted average of the coverages of every coverage_group instance.

The cov_weight attribute of a coverage_group determines the contribution of that group to the testbench coverage. The Coverage Attributes section in the *OpenVera LRM: Native Testbench* book describes coverage_group attributes and mechanisms for setting them.

The coverage for each coverage_group is the weighted sum of that group’s sample and cross coverage numbers. The cov_weight attribute of a sample determines the contribution of that sample to the coverage of the enclosing coverage group. Similarly, the cov_weight attribute of a cross determines the contribution of that cross to the coverage of the enclosing coverage group. Both attributes have a default value of 1. The Coverage Attributes section in the *OpenVera LRM: Native Testbench* book describes sample and cross attributes and mechanisms for setting them.

VCS computes the coverage number for a sample as the number of bins with the at_least number of hits divided by the total number of possible bins for the sample (multiplied by 100). When the sample is auto-binned (that is, there are no user-defined state or transition bins), the total number of possible bins for the sample is the minimum of the auto_bin_max attribute for that sample and the number of possible values for the coverage point.

By default, VCS does not create automatic bins for 'X' or 'Z' values of a coverage point. For example, if a coverage point is a 4 bit bit-vector and the `auto_bin_max` attribute is set to 64 (default), then by default the total number of possible bins for the coverage point is 16 (2^4). On the other hand, if VCS coverage is sampling the coverage point when it has 'X' or 'Z' values (`auto_bin_include_xz` attribute of the sample is set to ON or 1), then the total number of possible bins for the 4 bit bit-vector is 64 ($\text{MIN}(\text{auto_bin_max attribute}, 4^4)$). Finally, if the `auto_bin_max` attribute is set to 5, then the total number of possible bins for the 4 bit bit-vector is 5.

VCS computes the coverage number of a cross as the number of bins (of that cross) with the `at_least` number of hits divided by the total number of bins for that cross (multiplied by 100). By default, the number of possible bins for a cross is the sum of the user-defined bins and the number of possible automatically generated bins for that cross. The number of possible automatically generated bins is the product of the number of possible bins for each of the samples being crossed.

Reporting and Querying Coverage Numbers

Testbench coverage is reported in the coverage HTML and text reports (see [“Unified Coverage Reporting” on page 21-53](#) for details). The reports also include detailed information for each coverage group as well as the samples and crosses of each group.

You can also query for the testbench coverage during the VCS run. This allows you to react to the coverage statistics dynamically (for example, stop the VCS run when the testbench achieves a particular coverage).

The following system function returns the cumulative coverage (an integer between 0 and 100) for the testbench:

```
function integer get_coverage();
```

The following system function returns the instance-based coverage (an integer between -1 and 100) for the testbench:

```
function integer get_inst_coverage();
```

Note:

The `get_inst_coverage()` system function returns -1 when there is no instance-based coverage information (that is, the cumulative attribute of the `coverage_group` has not been set to 0).

See the *OpenVera LRM: Native Testbench* book for details on how to query for the coverage of individual sample and crosses of each `coverage_group` using the `query()` function.

Controlling Coverage Collection Globally

The `coverage_control()` system task (see the *OpenVera LRM: Native Testbench* for description of this task), coupled with the `-cg_coverage_control` runtime argument, provides a single-point mechanism for enabling/disabling of the coverage collection for all coverage groups or a particular coverage group.

Syntax

```
-cg_coverage_control=value
```

The values for `-cg_coverage_control` are 0 or 1. A value of 0 disables coverage collection, and a value of 1 enables coverage collection.

Example 21-2

```
#include <vera_defines.vrh>

coverage_group Cov{
    sample_event = @(posedge CLOCK);
    sample x {
        state x1(10);
        state x2(20);
        state x3(30);
        state x4(40);
        state x5(50);
        state x6(60);
        state x7(70);
    }
}

coverage_group Another{
    sample_event = @(posedge CLOCK);
    sample x{
        state x1(10);
        state x2(20);
        state x3(30);
        state x4(40);
        state x5(50);
        state x6(60);
        state x7(70);
    }
}

task query_and_print(string str){
    printf("Coverage is %d:%s\n",c.query(COVERAGE), str);
}

program test{
    integer x = 0;
    Cov c = new;
```

```
Another c1 = new;
@(posedge CLOCK);

coverage_control(0);

x = 10;
@(posedge CLOCK);

x = 30;
@(posedge CLOCK);

coverage_control(1);

x = 40;
@(posedge CLOCK);

x = 50;
@(posedge CLOCK);

coverage_control(0, "Cov");

x = 60;
@(posedge CLOCK);

coverage_control(1, "Cov");
coverage_control(0, "Another");

x = 70;
@(posedge CLOCK);
}
```

Unified Coverage Reporting

In VCS 2006.06, the db based coverage reporting has been replaced by the Unified Report Generator (URG) which can generate either HTML or text reports. The URG generates combined reports for all types of coverage information.

The format of the text report that the URG generates is different and better than the text report that used to be generated by the `ntb -cov_report` command line option. Any scripts that use these old command line options now need to be modified to use the URG options.

The functional coverage database files and their location have been changed. The coverage database is written to a top-level coverage directory. By default this directory name is `simv.vdb`. In general its name comes from the name of the executable file, with the `.vdb` extension. The reporting tool shipped with VCS version 2006.06 cannot read coverage databases generated using previous versions of VCS. Similarly, the reporting tool shipped with pre-VCS 2006.06 versions cannot read coverage databases generated using VCS 2006.06. The reports may be viewed through an overall summary "dashboard" for the entire design/testbench.

Coverage Reporting Flow

To generate coverage reports using URG, do the following:

1. Create the coverage database (for example, using VCS standalone):

```
% vcs -ntb test.vr
% simv
```

This runs a simulation and creates the directory `simv.vdb`

2. Create the coverage report from the database:

```
% urg -dir simv.vdb           //html
% urg -dir simv.vdb -format text //text
```

This creates the `urgReport` directory.

3. In order to view the results, invoke a browser (for example, invoke Mozilla):

```
% mozilla urgReport/dashboard.html  
% more urgReport/dashboard.txt
```

Please refer to the *Unified Report Generator User Guide* for details

Persistent Storage of Coverage Data and Post-Processing Tools

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 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 21-1 Unique Name Generation Scheme

Test Name	Database
pci_test	Database for the first testbench run.
pci_test_gen_1	Database for the second testbench run
pci_test_gen_2	Database for the 3rd testbench run
pci_test_gen_n	Database for the nth 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_file (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 Example 20-4 below, there is a Vera class `MyClass` 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 21-1

```
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 standalone 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 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 20-5, 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 21-2

```
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 over write the calls to `coverage_set_test_database_name` and loading 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, VCS automatically adds the extension `.vdb` to the directory name if not specified.

`-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".

Solver Choice

VCS incorporates two different solvers, each of which supports the entire OpenVera constraint language. You may select the solver you want using the runtime option `vera_solver_mode`, which takes the values 1 or 2. The default value is 2.

Example 21-3

```
simv +vera_solver_mode=2 ...
```

The two solvers have different performance characteristics. When the solver mode is set to 1, this solver exhaustively analyzes the entire solution space, restricted to the space allowed by the current assignment to non-random variables. It caches this analysis, and VCS reuses it in subsequent calls to `randomize()` with the same combination of non-random variables. Since the solver has a complete view of the solution space, it can generate a uniform sampling of the solutions.

When the solver mode is set to 2, this solver does an analysis of the solution space too, again restricted to the space allowed by the current assignment to non-random variables. However, it is not exhaustive as with solver mode 1. Consequently, the analysis is faster when the solver mode is set to 2. This solver then follows a heuristic search algorithm when it generates a random solution. Since this solver does not have a complete view of the solution space, it cannot generate a uniform sampling of the solutions.

Depending on the nature of constraints, and the number of calls to `randomize()` made per class, one or the other solver might be more suitable.

Automatic Solver Orchestration

As discussed before, VCS has two general purpose solvers. The user can indicate a preferred solver choice by setting the `+vera_solver_mode` option at runtime. In addition, there is a specialized solver that applies to simple constraint sets.

The system is very adaptive in the selection of the solver, and use the `+vera_solver_mode` setting is only interpreted as an initial guideline.

The behavior of the system is:

1. `+vera_solver_mode` remains the only user control provided. This setting will specify the preferred solver as either 1 or 2, which are the modes for the two general-purpose solvers mentioned above.
2. If an analysis of the constraints reveals that the specified constraints are better solved by the *non*-preferred solver, or by the specialized solver for simple constraints mentioned above, then the preferred solver setting is overridden. Thus, the initial solver choice is determined.
3. If the current solver choice results in a timeout while solving the constraints (timeout limits cannot be controlled by the user), then the solver mode is switched. A timeout cannot occur with the specialized solver, so the switched solver mode is either 1 or 2.
4. If the switched solver choice results in a timeout as well, then a timeout is reported to the user.

5. If the switched solver succeeds, then it remains the current solver choice.
6. Steps 3, 4, 5 are repeated for each call to randomize.

Temporal Assertions

OpenVera provides three pre-defined classes that enable the interaction of testbenches with OpenVera or SystemVerilog assertions. Instances of these classes (that is, the objects) are used to:

- Synchronize OpenVera threads with assertion engine events
- Synchronize OpenVera threads with individual assertion events
- Provide access to assertion properties

Table 21-2 OpenVera Assertion Classes

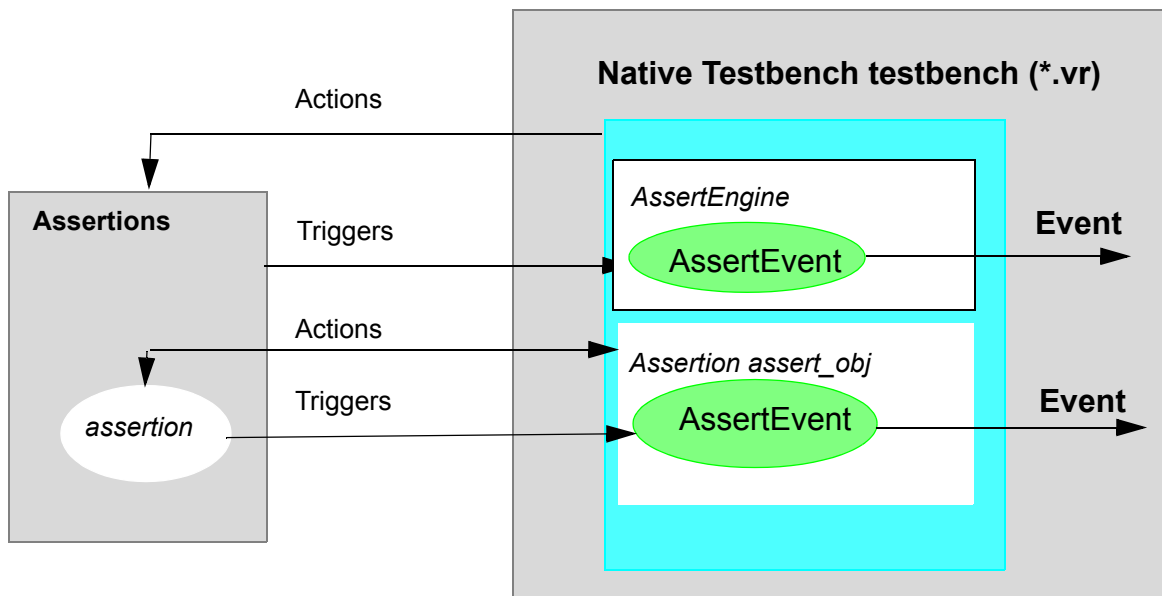
Object	Description
<code>AssertEngine</code>	Monitors and controls assertion as a whole: <ul style="list-style-type: none"> - Affects all assertions - Resets and disables all assertion attempts - Controls information displayed at runtime and in the report file
<code>Assertion</code>	Monitors and controls individual assertions: <ul style="list-style-type: none"> - Affects only a specific assertion - Resets and disables assertion attempts
<code>AssertEvent</code>	Synchronizes the testbench with events <ul style="list-style-type: none"> - Suspends a thread of the testbench

Figure 21-2 shows a typical verification runtime environment with the assertions.

Note:

In this chapter the term “assertion” refers to an OVA (OpenVera Assertion) or SVA (SystemVerilog Assertion) in the DUT or testbench. The class, “Assertion,” is the corresponding assertion object in the OpenVera program.

Figure 21-2 Interaction of Assertion Objects



Typically, you will have the following:

- Only one AssertEngine object
- An Assertion class object for every assertion used in the testbench or DUT
- An AssertEvent object for every event being monitored

Note:

This is not the event as defined in Assertions but rather as used in OpenVera. Here, an event is some occurrence that OpenVera can trigger on, such as a Failure or SUCCESS event.

For more information about Assertion class, see the section titled “OpenVera Temporal Assertion Classes” in Chapter 7 (“Predefined Methods and Procedures”) of the *OpenVera Language Reference Manual: Native Testbench*.

Temporal Assertion Flow

Once you have Assertions in the DUT or testbench, you may add Assertion classes to your OpenVera testbench in order to access or monitor the assertions. The following section explains how to do this, using an example testbench.

Note:

You may have OVA (OpenVera Assertion) or SVA (SystemVerilog Assertion) or SVA and OVA together, with Assertion classes in OpenVera.

Adding Assertion Objects to a Testbench

There are four major steps in adding assertion objects to a testbench:

1. Include the assertion.vrh file in the OpenVera code.
2. Create an AssertEngine object.
3. Create an Assertion object for assertion of interest in the DUT or testbench.

4. Create an AssertEvent object for each event to be monitored.

Including the Header Files

To start, every file that instantiates an Assertion object must include the header file:

```
#include <assertion.vrh>
```

Setting Up the AssertEngine Object

The program block must instantiate one, and only one, AssertEngine object. For a summary of the class and its methods, see the description of AssertEngine Class in the *OpenVera Language Reference Manual: Native Testbench* book.

```
program test {  
    ...  
    AssertEngine assert_engine = new();  
    ...  
}
```

Controlling Assertion Reporting

Once the engine is running, use its Configure() task to specify how the assertion reports results. The testbench settings take priority over compiler and runtime options.

Example 21-4 instantiates the AssertEngine, turns off the ASSERT_QUIET option, thus turning on runtime messages, and turns on line information in the messages with the ASSERT_INFO option. The ASSERT_REPORT option, which generates the ova.report file, is on by default.

Example 21-4

```
program {
    ...
    AssertEngine assert_engine = new();
    assert_engine.Configure(ASSERT_QUIET, ASSERT_FALSE);
    assert_engine.Configure(ASSERT_INFO, ASSERT_TRUE);
}
```

Resetting Assertion

At any time during the simulation, you can reset all attempts at matching assertions. You might want to reset when you switch to a new test and do not want to be confused by attempts started in the previous test. For example:

```
assert_engine.DoAction(ASSERT_RESET);
```

Instantiating Assertion Objects

Use AssertEngine functions to obtain handles to Assertion objects. You can specify the assertion or expression by the full hierarchical name. For example, for an assertion named `check1` in a unit named `test`:

```
Assertion assert_check1;
assert_check1 = assert_engine.GetAssert("test.check1");
```

Note:

When the assertion name is invalid, the value returned is “null.”

Another way is to create a loop and collect handles to all of the assertions. This way the number of assertions and their names can vary without changing the testbench. The `GetFirstAssert()` function can be used anytime to obtain the first assertion or expression in the compiled list. This is not necessarily the first in the Assertion source

file: the order might change while compiling. The `GetNextAssert()` function fetches more Assertion handles until it delivers a null, indicating the end of the list. You can identify the handles with the Assertion object's `GetName()` function.

In Example 21-5, the testbench sets up Assertion objects for all assertions and expressions, however many there might be:

Example 21-5

```
Assertion assert_check[];
string assert_name[];
int i = 0;
assert_check[0] = assert_engine.GetFirstAssert()
while (assert_check[i] != null)
{
    assert_name[i] = assert_check[i].GetName();
    ...
    assert_check[++i] = assert_engine.GetNextAssert();
}
```

This testbench also creates arrays of handles and names.

Controlling Evaluation Attempts

You can control attempts on individual assertions at any time during the simulation with the `DoAction()` task. For example, to reset attempts on the `test.check1` assertion shown in the example in the previous section, you could use the following:

```
assert_check1.DoAction(ASSERT_RESET);
```

Counting Successes and Failures

You can set up a running count of evaluation successes or failures. This does not need an `AssertEvent` object. To start the counting, just call the Assertion object's `EnableCount()` task. To see the current

value, call the object's `GetCount()` function. In Example 21-6, the code keeps track of how many times the assertion `check1` fails and generates an error, and ends the simulation, if there are too many failures.

Example 21-6

```
assert_check1.EnableCount(ASSERT_FAILURE);  
...  
if (assert_check1.GetCount(ASSERT_FAILURE) > max_fail)  
    error("Check1 exceeded failure limit.");
```

`EnableCount()` starts a running count that cannot be disabled or reset. To create a count that can be disabled or reset, set up an `AssertEvent` object (as explained in the following section) and create your own code to count the event triggers.

Setting Up the AssertEvent Objects

For every event being monitored, the testbench needs an `AssertEvent` object. For a summary of the class and its methods, see the description of the `Assertion` class in the *OpenVera LRM: Native Testbench book*.

Instantiating AssertEvent Objects

Instantiate an `AssertEvent` object with its `new()` task, including an event type. Then attach the object to an `AssertEngine` or `Assertion` object with that object's `EnableTrigger()` task. The event is not monitored until the `EnableTrigger()` task. For example, to monitor successful attempts on the `check1` assertion, use the following:

```
AssertEvent check1_success = new(ASSERT_SUCCESS);  
assert_check1.EnableTrigger(check1_success);
```

Or, to watch for a global reset by the AssertEngine, use the following:

```
AssertEvent assert_engine_reset = new(ASSERT_RESET);  
assert_engine.EnableTrigger(assert_engine_reset);
```

Each AssertEvent object can be used with only one AssertEngine or Assertion object. For example, to watch for ASSERT_SUCCESS on three assertions, you must create three AssertEvent objects and attach each one to a different assertion object.

Suspending Threads

AssertEvent objects are normally used to suspend a thread until the event happens. There are two ways to do this: the object's Wait() task or with Vera's sync() task and the object's Event variable. For example, using the Wait() task to wait for a global reset action:

```
assert_engine_reset.Wait();
```

After a thread resumes, you can see which events happened with the AssertEvent object's GetNextEvent() function. If called repeatedly, the function returns a list of events starting with the most recent and ending with ASSERT_NULL. For example:

```
reason = assert_engine_reset.GetNextEvent();
```

Eliminating AssertEvent Objects

When finished with an event, disable the trigger and recover the AssertEvent object's memory space with the associated AssertEngine or Assertion object's DisableTrigger() task.

Example 21-7

```
assert_engine.DisableTrigger(assert_engine_reset);
```


Terminating the AssertEngine

When the testbench is completely finished with assertions, it should terminate all assertion activity with the AssertEngine's DoAction() task. This task also recovers the memory space of all the assertion objects.

Example 21-8

```
assert_engine.DoAction(ASSERT_TERMINATE);
```

Example Testbench

The following complete testbench is provided as an example:

```
//Design under test
`timescale 1ns/100ps
module test;
    reg        clk;
    reg        dat;
    reg        syn;

    test_ntb vsh(
        .SystemClock ( clk )
    );

    initial begin
        clk = 0;
        #10 clk = 1;
        forever #5 clk = ~clk;
    end

    initial begin
        #2 dat = 0;
        #10 dat = 0;
        #10 dat = 0;
        #10 dat = 1;
        #10 dat = 0;
        #10 dat = 0;
        #10 dat = 1;
    end
endmodule
```

```

        #10 dat = 0;
        #10 dat = 1;
        #10 dat = 1;
        #10 dat = 0;
        #10 dat = 0;
        #10 dat = 0;
        $finish;
end

initial begin
    #2 syn = 0;
    #10 syn = 1;
    #10 syn = 0;
end
initial $monitor( "vcs: clk=%b dat=%b syn=%b time=%0d", clk,
    dat, syn, $time);
endmodule

//OpenVera
#include <assertion.vrh>
program test_ntb
{
    AssertEngine Eng = new();
    integer    nCycGbl = 0;

    fork
        AssertListener( "test.t1.ck1");
        AssertListener( "test.t1.ck2");
        AssertListener( "test.t1.ck3");
    join none

    DoCycles();
}

task
AssertListener(string tAsrt)
{

    Assertion Ast = Eng.GetAssert(tAsrt);
    AssertEvent Event = new( ASSERT_ALL );
    integer EventType;

```

```

printf( "\npgm: %s Assert.GetName() = %s " , tAsrt ,
        Ast.GetName());

Ast.EnableTrigger(Event);
printf("\npgm:Attached ASSERT_ALL event to %s " ,
        tAsrt);

while( 1)
{
    Event.Wait();

    EventType = Event.GetNextEvent();
    while( EventType != ASSERT_NULL )
    {
        printf( "pgm: Event.GetNextEvent():
                asrt=%s ev=%s\n",tAsrt,
                AssertGetType(EventType));
        EventType = Event.GetNextEvent();
    }
}
}
task
DoCycles()
{
    while( 1)
    {
        @( posedge CLOCK);
        nCycGbl++;
        printf( "pgm: time=%0d cyc=%0d \n", get_time(
                LO), nCycGbl);
    }
}
unit test_u (logic clk, logic dat, logic syn); //scope
test {

    clock posedge clk {
        event d0:  dat == 0  ;
        event d1:  dat == 1  ;
        event dsyn: d1 ->> d0 ->> d0 ->> d1 ->> d0 -
>> d1 ->>
                d1 ->> d0 ;
        event dk1 : if (syn == 1) then #1 dsyn ;

```

```
        event dk2 : if (syn == 1) then #2 dsyn ;
        event dk3 : if (syn == 1) then #3 dsyn ;
    }
    assert ck1: check(dk1);
    assert ck2: check(dk2);
    assert ck3: check(dk3);
endunit //}
bind instances test : test_u t1(clk,dat,syn);
```

Running OpenVera Testbench with OVA

Compilation:

```
vcs ov_options & ov_files design_files & vcs_options
ova_files ova_options
```

Assume that “design.v” is the DUT, “test.vr” is the OpenVera code containing assertion class, and the ova file is “checker.ova.”

Example 21-9

```
% vcs -ntb test.vr design.v checker.ova
```

Simulation:

```
% simv runtime_ova_options runtime_NTB_option
% simv
```

Running OpenVera Testbench with SVA

Compilation:

```
% vcs ov_options & ov_files sva_files_sva_options +sysvcs
```

Simulation:

```
% simv runtime_sva_options
```

Running OpenVera Testbench with SVA and OVA Together

Compilation:

```
vcs ov_options_&_ov_files ova_&_ova_options \  
    sva_options_&_sva_files +sysvcs
```

Simulation:

```
simv simv_options
```

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 section describes:

- The Scope of Interoperability
- Using the SystemVerilog package import syntax to import OpenVera data types and constructs into SystemVerilog.
- Calling of OpenVera tasks, functions and methods from SystemVerilog. Tasks and functions can be imported to SystemVerilog using the same method for importing classes.

- 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).
- Working with synchronization objects such as events, mailboxes and semaphores across the language boundary.
- Mapping of SystemVerilog modports to OpenVera where they can be used as OpenVera virtual ports.
- Effect of directly or indirectly calling blocking OpenVera functions from SystemVerilog.
- Handling of differences in the semantics of sample, drive, expect, etc. between OpenVera and SystemVerilog.
- The OpenVera-SystemVerilog interoperability use model.

Scope of Interoperability

The scope of OpenVera-SystemVerilog interoperability in VCS Native Testbench is as follows:

- Classes defined in OpenVera, that you use directly or extend in a SystemVerilog testbench
- A testbench in SystemVerilog. The testbench uses SystemVerilog constructs like interfaces with modports, virtual interfaces with modports, and clocking blocks to communicate with the design.
- OpenVera code does not contain program blocks, interfaces, bind statements, classes, enums, ports, tasks and functions.
- Your OpenVera code uses 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 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 an 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

This does not however 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 above example would be valid if we add the following line before the first usage of the name `Base`.

```
import OpenVera::Base;
```

Continuing 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 the above 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

In this section, we describe 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 plans to support the mapping of mailboxes between the two languages as follows:

Consider a mailbox created in SystemVerilog. To use it in OpenVera, we 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, we 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 this 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 table below.

SystemVerilog	OpenVera	2/4 or 4/2 value conversion?	Change in signedness?
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.

Of additional interest is the reverse map for the OV bit type:

OV	SV	2/4 or 4/2 value conversion?
change in signedness?	signedness?	
4/2		conv?
bit[m:n]	logic[m:n]N	N

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:

OpenVera	SystemVerilog	Compatible?
<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

Note:

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 can't 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

bit clk;
    :
IFC i(clk);
    :
virtual IFC.mp vmp;
    :
    vmp = i.mp;
    @(vmp.cb.a); // here we need to specify cb explicitly
```

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
```

```

    bit clk;
    :
    IFC i(clk);
    :
    virtual IFC.mp vmp;
    :
        vmp = i.mp;
        @(vmp.a); // cb can be omitted; 'cb.a' is
                // imported into the modport

// 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
bit clk;
:
IFC i(clk);
:
virtual IFC.mp vmp;
:
    vmp = i.mp;
    @(vmp.a); // cb can be omitted;
            // 'cb.a' is imported into the modport

```

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 usage:

```
// 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 cb @(posedge clk);
        output a;
        input b;
    endclocking
    modport mp (clocking cb, import cb.*);
endinterface: IFC

import OpenVera::Foo;

:
IFC ifc(clk); // use this to connect to DUT and TB
:
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();
:

```

Note:

It is not necessary to use a *virtual* modport above. One can directly pass a modport from an instance of an interface as follows:

```

Foo f = new(ifc.mp);

```

A modport can aggregate signals from multiple clocking blocks.

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`.

Miscellaneous Issues

Blocking Functions in OpenVera

When a SystemVerilog function calls a virtual function that may resolve to a blocking OpenVera function at run-time, 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 run-time

The `terminate`, `wait_child`, `disable fork`, and `wait fork` Constructs

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 Svobj=new();
SVobj.randomize();
obj.randomize() with
{obj.ri==SVobj.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 ccl(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

```

Use 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.

Support for multiple program blocks in OV and SV is not present at this time.

VCS compile:

```
vcs -sverilog -ntb_opts interop +Other_NTB_options
-ntb_incdir incdir1+incdir2+...
-ntb_define defines_for_OV_source_files
OpenVera_.vrp_files
OpenVera_source_files_.vr
SystemVerilog_source_files_and_libraries
```

A few more compile options are significant:

1. if RVM class libs are used in the OV code, this is required:

```
-ntb_opts rvm
```

2. VMM classes, vmm_ macros can be used in SV; and rvm_ macros in OV package are automatically translated to vmm_ equivalents if this is also added

```
-ntb_opts interop -ntb_opts rvm
```

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 Vera to 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.

Using Reference Verification Methodology with OpenVera

VCS supports the use of Reference Verification Methodology (RVM) for implementing testbenches as part of a scalable verification architecture.

The syntax for using RVM with VCS is:

```
vcs -ntb_opts rvm [vcs_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 non-tool specific descriptions apply to NTB.

Differences between use 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 switch `-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, no guarantee of fairness in reads from channels, etc.

Testbench Optimization

VCS/VCSi supports a set of features to analyze a testbench, identifying potential areas of improvement, and to partition the simulation run, reducing overall simulation time.

NTB 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 in conjunction with `-ntb` and other NTB options when compiling the file. For example:

```
% vcs -ntb +prof other_ntb_compile_time_options \  
    verilog_files testbench_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. Program `MyTest` calls an OpenVera task `MyPack`, and a DPI task `DPI_call`, together in a loop 20 times. The profiler reports the relative portion of the runtime that each consumes.

Example 21-10

```
#include <vera_defines.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)
    {
        @(posedge CLOCK);
        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
    }
    @(posedge CLOCK);
}

```

Example 21-11 *dpi.c*

```

#include <svdpi.h>

static int tmp;
static void do_activity(int j)
{
    int i;

    for( i = 0; i < 1000; 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 );
    }
}

```

Compile:

```
% vcs -R -ntb +prof dpi.vr dpi.c
```

Run:

```
% ./simv
```

Example 21-12 Log File (vcs.prof)

```

// Synopsys VCS X-2005.09-A[D] (ENG)
// Simulation profile: vcs.prof
// Simulation Time:      8.360 seconds

```

=====

TOP LEVEL VIEW

TYPE	%Totaltime
DPI	98.60
PLI	0.00
VCD	0.00
KERNEL	0.00
MODULES	0.00
PROGRAMS	1.40

PROGRAM VIEW

Program(index)	%Totaltime	No of Instances	Definition
----------------	------------	-----------------	------------

MyTest (1) 1.40 1 test.vr:27.

PROGRAM TO CONSTRUCT MAPPING

1. MyTest

Construct	Construct type	%Totaltime	%Programtime	LineNo
Fork	Program Thread	0.84	60.00	test.vr : 39-42.
A::pack	Object Pack	0.42	30.00	test.vr : 24.
MyPack	Program Task	0.14	10.00	test.vr : 12-24.

TOP-LEVEL CONSTRUCT VIEW

Construct	%Totaltime
Program Thread	0.84
Object Pack	0.42
Program Task	0.14

CONSTRUCT VIEW ACROSS DESIGN

1. Program Thread

Program	%Totaltime
MyTest	0.84

2. Object Pack

Program	%Totaltime
MyTest	0.42

3. Program Task

```

-----
                Program          %Totaltime
-----
                MyTest          0.14
-----
-----
// Simulation memory: 3561082 bytes
-----
-----
                TOP LEVEL VIEW
-----
-----
                Type          Memory          %Totalmemory
-----
-----
                DPI          1024          0.03
                PLI          0          0.00
                VCD          0          0.00
                KERNEL      1414243          39.71
                MODULES     0          0.00
                PROGRAMS    2145815          60.26
-----
-----
                PROGRAM VIEW
-----
-----
Program(index) Memory %Totaltime No of Instances Definition
-----
*****End of vcs.prof*****

```

The vcs.prof log file shown in Example 21-12 provides the following information.

- The DPI function in dpi.c consumed 98.6% of the total time.

Fork	Program Thread	0.84	60.00	test.vr : 39-42.
A::pack	Object Pack	0.42	30.00	test.vr : 24.
MyPack	Program Task	0.14	10.00	test.vr : 12-24.

- The fork-join block defined in test.vr:39-42 consumed 0.84% of the total time.

- The predefined class method, `pack()`, invoked at `test.vr:24` consumed 0.42% of the total time.
- The task `MyPack()` defined at `test.vr:12:24` consumed 0.14% 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.

VCS Memory Profiler

The VCS memory profiler is an Limited Customer Availability (LCA) feature in VCS and requires a separate license. Please contact your Synopsys AC for a license key.

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.

Use Model

The `$vcsmemprof()` task can be called from the CLI or the UCLI interface. The syntax for `$vcsmemprof()` is as follows:

```
$vcsmemprof("filename", "w|a");
```

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 -ntb [-sverilog] +dmprof dut_filename.v  
testbench_filename.vr \[-debug | -debug_all]
```

Note:

Use the `-sverilog` compile-time option when compiling SystemVerilog code. OpenVera code does not require this option.

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")}
```

CLI Interface

Compile-time

The dynamic memory profiler is enabled only if you specify `+dmprof` on the VCS compile-time command line:

```
vcs -ntb [-sverilog] +dmprof dut_filename.v testbench_filename.vr \  
        [-debug | -debug_all]
```

Note:

Use the `-sverilog` compile-time option when compiling SystemVerilog code. OpenVera code does not require this option.

Runtime

At runtime, invoke `$vcsmemprof()` from the CLI command line prompt as follows. You can make the call to `$vcsmemprof()` at any point during the simulation.

```
simv -s //Invokes the cli prompt  
cli_0>$vcsmemprof("memprof.txt", "w|a+")
```

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.

Incremental Profiling

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.

Only Active Memory Reported

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.

VCS Dynamic Memory Profile Report

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

a. Task-Function-Thread section

Reports the total dynamic memory in each active task, function and thread in the module/program.

b. Class Section

Reports the total dynamic memory consumed in each class in the module/program.

c. Dynamic data Section

Reports the total memory consumed in each of 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 21-13

```
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);
    }

```


Compile:

```
vcs -ntb +dmprof test.vr -debug_all
```

Note:

The `-sverilog` compile-time option is not used, since the program involves OpenVera code.

Run:

```
simv -ucli // Invokes ucli prompt
ucli> next
ucli> next
ucli>call {$vcsmemprof("memprof.txt", "w")} // "w" the mode the file is
// opened in.
```

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 Defination
              Memory      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      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

22

SystemVerilog Design Constructs

This chapter begins with examples of the constructs in SystemVerilog that VCS has implemented. It then describes enabling the use of SystemVerilog code.

This chapter covers the following topics:

- [SystemVerilog Data Types](#)
- [Writing To Variables](#)
- [SystemVerilog Operators](#)
- [New Procedural Statements](#)
- [SystemVerilog Processes](#)
- [Tasks and Functions](#)
- [Hierarchy](#)

- [Interfaces](#)
- [Enabling SystemVerilog](#)
- [Disabling unique And priority Warning Messages](#)

SystemVerilog Data Types

SystemVerilog has several new data types which are described in the following sections.

Variable Data Types for Storing Integers

VCS has implemented the following SystemVerilog variable data types for storing integers:

data type	States	Default	Description
<code>char</code>	two state	signed	A C-like data type, 8-bit integer
<code>shortint</code>	two state	signed	16-bit integer
<code>int</code>	two state	signed	32-bit integer
<code>longint</code>	two state	signed	64-bit integer
<code>byte</code>	two state	signed	8-bit integer or ASCII character
<code>bit</code>	two state	unsigned	User-defined vector size
<code>logic</code>	four state	unsigned	User-defined vector size

Notice that some of these data types default to signed values. The Verilog-2001 standard has the `unsigned` reserved keyword. In SystemVerilog you can use it in the variable declaration to change one of these default signed data types to unsigned. For example:

```
longint unsigned liu;
```

You can also use the `signed` keyword to make the `bit` and `logic` data types store signed values.

Note:

In SystemVerilog the Verilog-2001 `integer` data type defaults to a signed value.

The two state data types begin simulation with the 0 value. Assignments of the Z or X values to these data types result in an assignment of the 0 value.

There is also the `void` data type that represents non-existent data. This data type can be used to specify that a function has no return value. For example:

```
function void nfunc (bit [31:0] ia);
:
endfunction
```

The chandle Data Type

The chandle data type is for pointers that you pass using the DPI.

The DPI and the chandle data type are LCA features requiring a special license.

The following example shows the use of the chandle data type:

```
program Test(input clk);

typedef struct packed {
    time t;
    int packet_id;
    // more stuff
} Transaction;
```

```

typedefchandle DataBase; // Data Base implemented in C/C++

import "DPI" function DataBase openDataBase(string name);
import "DPI" function void saveDataBase(DataBase db);
import "DPI" function void addToDataBase(DataBase db,
Transaction tr);

DataBase myDataBase;

initial begin
    myDataBase = openDataBase("TestResults_1");
    // ...
    while (1) begin
        // process transactions
        Transaction tr;
        // ...
        // record the just processed transaction in a data
base
        addToDataBase(myDataBase, tr);
        // ...
    end
end

// ...

final begin
    saveDataBase(myDataBase);
end
endprogram

```

This example uses achandle to refer transactions to a C++ database.

User-Defined Data Types

You can define your own data types from other data types using the `typedef` construct like in C. For example:

```
typedef logic [63:0] mylog;
mylog m1, m2, m3, m4;
```

Following the `typedef` keyword are the SystemVerilog data type for the user-defined type, the optional bit-width, and the name of the user-defined type.

You can use a user-defined type, that is, declare a signal that has a user-defined type, before the definition of the user-defined type. For example:

```
typedef mine;
mine p;
:
typedef int mine;
```

You enable this use of a user-defined type by entering the `typedef` keyword and the name of the user-defined type without the SystemVerilog data type.

Enumerations

You can declare a set of variables that have a set of values. These are called enumerated data types. For example:

```
enum shortint{green,yellow,red} northsouth,eastwest;
```

In this example we have declared `shortint` variables `northsouth` and `eastwest`, that can hold a set of unassigned constants named `green`, `yellow`, and `red`.

The default data type for an enumerated data type is `int`. If you omit the data type in the enumeration, the variables declared have the `int` data type.

You can make assignments to the constants in the enumeration.

For example:

```
enum{now=0,next=1,old=2}cs,ns,tmp;
```

You can declare a user-defined data type and then use it as the base type for an enumeration. For example:

```
typedef bit [1:0] mybit;
typedef enum mybit {red=2'b00, green=2'b01, blue=2'b10,
                  yellow=2'b11} colors;
```

You can use the following data types as the base type of an enumeration:

```
reg logic int integer shortint longint byte
```

Unpacked dimensions are not allowed in the base type.

Methods for Enumerations

SystemVerilog contains a number of methods for enumerations. These methods do the following:

`.first`

Displays the value of the first constant of the enumeration.

`.last`

Displays the value of the last constant of the enumeration.

`.next`

Displays the value of the next constant of the enumeration.

`.prev`

Displays the value of the previous constant in the enumeration.

`.num`

Displays the total number of constants in the enumeration.

`.name`

Displays the name of the constant in the enumeration.

The following is an example of the use of these methods:

```
module top;
typedef enum {red, green, blue, yellow} Colors;
```

Here is an enumeration named Colors. It has four constants named red, green, blue and yellow and ranging in value from 0 to 3.

```
Colors color = color.first;
```

We declare a Colors variable named color and initialize it to the value of the first constant in the enumeration.

```
initial
begin
$display("Type Colors:\n");
$display("name\tvalue\ttotal\tprevious\tnext\tlast\tfirst\n");
forever begin
  $display("%0s\t%0d\t%0d\t%0d\t\t%0d\t%0d\t%0d\n",
    color.name,color,color.num,color.prev,color.next,color.last,
    color.first);
  if (color == color.last) break;
  color = color.next;
end
end
```

```
endmodule
```

Results from the VCS simulation are as follows:

Type Colors:

name	value	total	previous	next	last	first
red	0	4	3	1	3	0
green	1	4	0	2	3	0
blue	2	4	1	3	3	0
yellow	3	4	2	0	3	0

The \$typeof System Function

The `$typeof` SystemVerilog system function returns the data type of its argument, and its argument can be either of the following:

- A primary expression. In this case a primary expression is a net or variable, bit or part select of a net or variable, or a member of a structure or union.
- A data type

The expression cannot be a cross module reference such as a hierarchal name of a signal outside of the module definition or an element in a dynamic array.

The actual returned value is not accessible to you. You can't display the value. You use the returned value when comparing the data types of various signals.

The following are some examples of the use of this system function:

```

module test;
logic [31:0] log1,log2;
bit [7:0] bit1;
parameter type bus_t = $typeof(log1);

```

In this type parameter, `bus_t` gets the data type of `log1`, which is `logic[31:0]`.

```

initial
begin
if ($typeof(log1) == $typeof(log2))
    $display("log1 same data type as log2");
if ($typeof(log1) != $typeof(bit1));
    $display("log1 not the same data type as bit1");
if ($typeof(log2) == $typeof(logic [31:0]))
    $display("log2 is logic [31:0]");

```

VCS executes all three of these `$display` system tasks. Notice that the argument to the second `$typeof` system function, in the last if statement, is not a signal name but a data type. This is the way of determining if a signal's data type is a specified data type.

```

end
endmodule

```

Be advised that you cannot use this system function to supply a data type at the start of a signal declaration. The following is invalid:

```

$typeof(log1) log2;

```

Specifying the wrong kind of argument results in the following error message:

```

Error-[SVFNYYI] System Verilog feature not yet implemented
    $typeof system function is currently supported only for
    primary expressions, selects and slices on primary
    expressions, structure member references and data types.

```

Expression: *expression*
"filename", *line_number*

Structures and Unions

You can declare C-like structures and unions. The following are some examples of structures:

```
struct { logic [31:0] lg1; bit [7:0] bt1; } st1;
struct {
    bit [2:0] bt2;
    struct{
        shortint st1;
        longint lg1;
    } st2;
} st3;
```

In these three structures:

- The first structure is named `st1` and it contains a 32 bit logic variable named `lg1` and an 8 bit `bit` variable named `bt1`.
- The second structure named `st3` contains a structure named `st2`. Structure `st3` also contains a `bit` variable named `bt2`.
- Structure `st2` contains a `shortint` variable named `st1` (same name as the first structure, this is okay because structure `st1` is not in the same hierarchy as `shortint st1` but it is not recommended). Structure `st2` contains a `longint` variable named `lg1`. Notice there is also an `lg1` in structure `st1`.

You can assign values to and from the members of these structures using the structure names as the building blocks of hierarchical names. For example:

```
initial
begin
```

```
logic1 = st1.lg1;
longint1 = st3.st2.lg1;
st1.bt1 = bit1;
end
```

You can make assignments to an entire structure if you also make the structure a user-defined type. For example:

```
typedef struct { logic [7:0] lg2; bit [2:0] bit2;} st4;
st4 st4_1;

initial
st4_1={128,3};
```

The keyword `typedef` makes the structure a user-defined type. Then we can declare a variable of that type. Then we can assign values to the members of the structure. Here `lg2` is assigned 128 and `bit2` is assigned 3.

A structure can be packed or unpacked. A packed structure is packed in memory without gaps so that its members represent bit or part selects of a single vector. This isn't true with unpacked structures. You specify a packed structure with the `packed` keyword. By default structures are unpacked. The following is an example of a packed structure:

```
struct packed { bit [15:0] left; bit [7:0] right;} lr;
```

With a packed structure you can access the values of the members by accessing bit or part selects of the packed structure's vector. For example:

```
initial
begin
lr.left='1;
lr.right='0;
```

```

mbit1=1r[23:8];
mbit2=1r[7:0];
$display("mbit1=%0b mbit2=%0b",mbit1,mbit2);
end

```

In SystemVerilog you can enter an unsized literal single bit preceded by an apostrophe ' as shown in the assignments to members left and right. All bits the variable are assigned the unsized literals single bit. In this case left is filled with ones and right is filled with zeroes.

Here the \$display system task displays the following:

```

mbit1=1111111111111111 mbit2=0

```

VCS has not implemented an unpacked union. In a packed union all members must be packed structures, packed arrays (see ["SystemVerilog Arrays" on page 22-14](#)), or integer data types, all with the same size, for example:

```

typedef struct packed { bit [15:0] b1; bit [7:0] b2;} st24;
typedef union packed{
    st24 st24_1;
    reg [23:0] r1;
    reg [7:0][2:0] r2;
} union1;

```

In this union all the members, structure st24, reg r1, and reg r2 have 24 bits.

You can access the members of the union as follows:

```

union1 u1;
bit [7:0] mybit1;
reg [7:0]myreg1;
initial
begin
mybit1=u1.st24_1.b2;

```



```
myreg1=u1.r2[2];  
end
```

Structure Expressions

You can use braces and commas to specify an expression to assign values to the members of a structure. For example:

```
typedef struct{  
    bit bt0;  
    bit bt11;  
} struct0;  
struct0 s0 = {0,1};
```

In this example, in the declaration of struct0 structure s0, bt0 is initialized to 0 and bt11 is initialized to 1, because they are listed that way in the declaration of structure struct0.

You can use the names of the members in a structure expression so that you do not have to assign values in the order of the members in the declaration. For example:

```
typedef struct{  
    bit bt1;  
    bit bt2;  
} struct1;  
struct1 s1 = {bt2:0, bt1:1};
```

You can use the default keyword to assign values to unspecified members. You can also use a structure expression in a procedural assignment statement. For example:

```
typedef struct{  
    logic l1;  
    bit bt3;  
}struct2;  
struct2 s2;
```

```
initial
s2 = {default:0};
```

SystemVerilog Arrays

SystemVerilog has packed and unpacked arrays. In a packed array all the dimensions are specified to the left of the variable name. In an unpacked array the dimensions are to the right of the variable name. For example:

```
bit [1:0] b1;           // packed array
bit signed [10:0] b2;  // packed array
logic l1 [31:0];       // unpacked array
```

Packed arrays can only have the following variable data types: `bit`, `logic`, and `reg`. You can make a packed array of any net data type.

Unpacked arrays can be made of any data type.

When assigning to and from an unpacked array the following rules must be followed:

- You cannot assign to them an integer, for example:

```
logic l1 [31:0];       // unpacked array
:
l1 = '0;
```

- You cannot treat them as an integer in an expression, for example:

```
logic l1 [31:0];       // unpacked array
:
reg2 = (l1 + 2);
```

- You can only assign another unpacked array with the same number of dimensions, all with the same size.

When assigning to a packed array you can assign any vector expression, for example:

```
bit [1:0] b1;           // packed array
bit signed [10:0] b2;  // packed array
logic [7:0] l2;        // packed array
:
b1={r1,r2};
b2='1;
l2=b2[7:0];
```

Multiple Dimensions

You can have multi-dimensional arrays where all the dimensions are packed or some dimensions are packed and others unpacked. Here is an example of all dimensions packed:

```
logic [7:0][3:0][9:0] log1;
```

Here, is a single entry of forty bytes. All dimensions are packed, so in an assignment to this array, you reference the dimensions from left to right. To assign to the left-most bit in the left most dimensions, do the following:

```
log1[7][3][9]=1;
```

Here is an example of none of the dimensions packed:

```
logic log2 [15:0][1:0][4:0];
```

Here are ten entries of two bytes. Like when all dimensions are packed, when all dimensions are unpacked, in an assignment to this array, you reference the dimensions from left to right. To assign to the left-most bit in the left most dimensions, do the following:

```
log2[15][1][4]=1;
```

Here is an example in which some dimensions are packed, but another is not:

```
logic [11:0][1:0] log3 [6:0];
```

Here are seven entries of three bytes. In an assignment to this array, you reference the unpacked dimensions, followed by the packed ones. To assign to the left-most bit in the left most dimensions, do the following:

```
log3[6][11][1]=1;
```

In these assignments the last packed dimension can be a part select, or a slice. For example:

```
log3[6][11][1:0]=2'b11;  
log3[6][11:10]=2'b00;
```

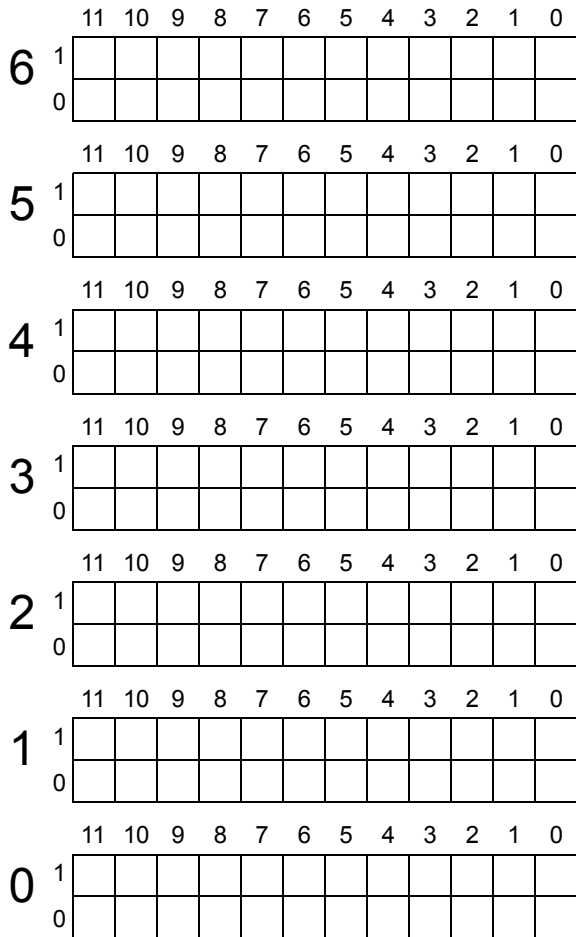
Indexing and Slicing Arrays

SystemVerilog has both part-selects and slices. To illustrate this concept, consider the last example multidimensional array:

```
logic [11:0][1:0] log3 [6:0];
```

A graphic representation of the packed and unpacked dimensions of this array is in Figure 22-1.

Figure 22-1 Packed and Unpacked Multi-dimensional Array



The first of the previous two assignment statements:

```
log3[6][11][1:0]=2'b11;
```

This is an assignment to a part select. It is an assignment to contiguous bits on a single packed dimension.

Figure 22-2 A SystemVerilog Part Select

		11	10	9	8	7	6	5	4	3	2	1	0
6	1	1											
	0	1											

The second of the two previous assignment statements:

```
log3[6][11:10]=2'b00;
```

This is an assignment to a slice. A slice is a joint part select in contiguous elements of a multidimensional array. This slice assignment is shown in Figure 22-3.

Figure 22-3 A SystemVerilog Slice

		11	10	9	8	7	6	5	4	3	2	1	0
6	1	0	0										
	0	0	0										

SystemVerilog Testbench Constructs Outside Programs

SystemVerilog testbench constructs outside programs, for example in modules, packages and in \$root, is an LCA feature requiring a special license. SystemVerilog packages are described in ["SystemVerilog Packages" on page 22-46](#).

You enable testbench constructs outside programs with the `-ntb_opts dtm` compile-time option and keyword argument. The keyword name comes from “dynamic types in modules.”

The testbench constructs that you can enter outside programs with this option are as follows:

classes associative arrays dynamic arrays
SystemVerilog named events

For descriptions and examples of these constructs see Chapter 24, "SystemVerilog Testbench Constructs".

Writing To Variables

SystemVerilog changes one of the basic concepts of Verilog: that variables are only written to by procedural statements. In SystemVerilog there are several other ways to write to a variable, as the following code example illustrates:

```
module dat;
logic log1,log2,log3;
longint loil;
byte byt1;
wire w1;

assign log1=w1;          //continuous assignment to logic
assign loil=w1;         //continuous assignment to longint
assign byt1=w1;        //continuous assignment to byte
buf b1 (log2,w1);      //connect to output terminal
dev dev1(log3,w1);     //connect to output port
endmodule

module dev(output out,input in);
assign out=in;
endmodule
```

As you can see, in SystemVerilog, you can write to a variable using a continuous assignment, connecting it to an output terminal of a primitive, or an output port of a module instance. Doing so is called using a structural driver.

There are some limitations on structural drivers on variables:

- A variable cannot have a behavioral driver (assigned a value by a procedural assignment statement) and a structural driver.
- A variable, unlike a net, cannot have multiple structural drivers.
- A variable still cannot connect to an `inout` port of a module definition.

Note:

You can also declare a variable to be an `input` port, this is also using a structural driver, See ["New Port Connection Rules for Variables" on page 22-59](#).

Force and Release on SystemVerilog Variables

You can enter `force` and `release` statements for SystemVerilog data types. The following SystemVerilog code declares such data types and `force` and `release` statements for them:

```
module test;
  int int1,int2;

  initial
  begin
    force int1=100;
    force int2=100;
    #100 release int1;
    release int2;
  end
```



```
endmodule
```

Automatic Variables

You cannot force a value on to an automatic variable. For example:

```
task automatic my_aut_task;
:
begin
:
#1 force mat=1; // causes this warning:
:
end
endtask
```

Doing so makes the task static and VCS displays the following warning message:

```
Warning-[MNA] Making Task or Function Non-Automatic
Disable/Force/Release/Assign/Deassign inside Automatic Task
is not supported.
  "filename.v", line_number:
  task automatic task_name;
```

Also any cross-reference to or from an automatic variable makes the task static. This happens with a `force` statement, but also with any kind of assignment statement. All of the assignment statements and `force` statements in the following code result in VCS compiling the task as a static task:

```
initial
begin
#5 r5=my_aut_task.mat;
#5 my_aut_task.mat =1;
#5 force my_aut_task.mat=1;
#5 force r5=my_aut_task.mat;
end
```

VCS displays the following warning message:

```
Warning-[MNA] Making Task or Function Non-Automatic
Making Task non-automatic due to Xmr access into it.
"filename.v", line_number:
task automatic task_name;
```

Multiple Drivers

If different elements of an unpacked array have structural and procedural drivers, then you cannot enter a `force` statement for any of the elements in the unpacked array.

A procedural driver is a procedural assignment statement.

A structural driver is, for example, a continuous assignment or a connection to an output or inout port in a module instance.

```
module mult_driver;
int unpacked_int [3:0];
assign unpacked_int[3]=1; // structural driver
dev dev1(unpacked_int[3],1); // structural driver

initial
begin
unpacked_int[2]=1; // procedural driver
//force unpacked_int[1]=1; //force statement is not valid
end
endmodule

module dev(out,in);
output [31:0] out;
input [31:0] in;
assign out=in;
endmodule
```

The `force` statement causes the following error message:

Error-[IFRLHS] Illegal force/release left hand side
Force/Release of an unpacked array which is driven by a mixture of structural and procedural assignments is not valid

The offending expression is *signal_name*
"mult_driver.v", 10: force *signal_name*[1] = 1;

This restriction does not hold for unpacked structures, see ["Structures" on page 22-27](#).

Release Behavior

When VCS executes a `release` statement on a SystemVerilog variable, the release behavior depends on how that variable obtained the value it had before VCS executed the previous `force` statement on it.

- If it was a structural driver, which is to say a connection to a module or primitive instance or by a continuous assignment or a procedural continuous assignment, the variable returns to its previous value immediately.
- If it was a behavioral driver, otherwise known as a procedural assignment statement, the variable returns to that previous value when or if VCS executes the procedural assignment statement again. Until VCS executes the assignment again, the variable keeps its forced value.

The following SystemVerilog module illustrates this behavior:

```
module test;
logic l1, l2, l3;
:
assign #10 l1=1;
:
always @(posedge clk)
```

```

begin
assign l2=1;
l3=1;
#10 force l1=0;
      force l2=0;
      force l3=0;
#10 release l1;
      release l2;
      release l3;
end
endmodule

```

Signal l1 is continuously assigned a value of 1 at simulation time 10. Assuming a rising edge on the clock signal at simulation time 100, a procedural continuous assignment assigns the value of 1 to signal l2 and a procedural assignment assigns the value of 1 to signal l3.

At simulation time 110, all three signals are forced to a 0 value.

At simulation time 120, when VCS executes all three `release` statements, signals l1 and l2 return to their 1 values, but signal l3 remains at 0 until the next rising edge on the clock.

Integer Data Types

The SystemVerilog LRM lists the following data types as integer data types:

```

shortint int longint byte bit logic reg integer time

```

All of these data types can hold an integer value. With the exception of the `time` data type, you can force a value on to an entire signal with these data types or a bit-select or part-select of these data types.

The following are examples of these data type declarations and valid `force` statements to the most significant bits in them:

```
shortint si1;
int int1;
longint li1;
byte byt1;
bit [31:0] bit1;
logic [31:0] log1;
reg [31:0] reg1;
integer intg1;

initial
begin
force si1[15]=1;
force int1[31]=1;
force li1[63]=1;
force byt1[7]=1;
force bit1[31]=1;
force log1[31]=1;
force reg1[31]=1;
force intg1[31]=1;
end
```

Notice that a bit-width was not specified for the `shortint`, `int`, `longint`, `byte`, or `integer` data types in their declarations, but these `force` statements to these bit-selects are valid. This is because these data types have a known number of bits, with little-endian numbering.

Force statements to bit or part-selects of the `real`, `time`, or `realtime` data types are not possible.

Unpacked Arrays

You can make `force` statements to an element of an unpacked array or a slice of elements. For example:

```
int int1;
logic l1 [2:0];
bit bit1 [2:0];
reg reg1 [2:0];
byte byte1 [2:0];
int int2 [2:0];
shortint si1[2:0];
longint li1[2:0];
integer intg1[2:0];

initial
begin
int1=2;
force l1[int1]=3;
force bit1[int1]=3;
force reg1[int1]=3;
force byte1[int1]=3;
force int2[int1]=3;
force si1[int1]=3;
force li1[int1]=3;
force intg1[int1]=3;
end
```

You can force an entire unpacked array, a slice of elements, or a single element. For example:

```
int int1 [2:0];
int int2 [2:0];

initial
begin
force int1={1,1,1};
force int2[2:1]=int1[2:1];
force int2[0]=0;
end
```

Like with `force` statements, you can enter a `release` statement for an entire array, a slice of elements in the array, or a single element.

You can use a variable to specify the element of an unpacked array. For example:

```
int int1;
logic l1 [2:0];

initial
begin
int1=2;
force l1[int1]=3;
end
```

If however, you have an unpacked array of packed bits, the packed bits must be specified with a value or a parameter, not a variable. For example:

```
int int1=1;
parameter p1=4;
logic log1 [2:0];
logic [5:3]log2[2:0];

initial
begin
force log1[int1]=1;
force log2[int1][p1]=1;
end
```

The `const` constant construct will not work for specifying packed bits.

Structures

You can force and release both entire packed and unpacked structures, or individual members of the structure. For example:

```

typedef struct{
    int int1;
    bit [31:0] packedbit;
    integer intg1;
} unpackedstruct;

typedef struct packed{
    logic [2:0] log1;
    bit [2:0] bit1;
}packedstruct;

module test;

unpackedstruct ups1;
unpackedstruct ups2;
packedstruct ps1;
packedstruct ps2;

assign ups1.int1=30;

initial
begin
    #0 ups1.packedbit[0]=0;
    #20 force ups1.int1=1;
    #20 force ups2=ups1;
    #20 release ups2.int1;
    #20 release ups2;
    #20 force ps1.log1=1;
    #20 force ps1.bit1=0;
    #20 force ps2=ps1;
    #20 release ps1;
end
endmodule

```

Using the VPI

You can force and release on SystemVerilog variables using the VPI. The following conditions apply:

- VPI force and release should behave exactly in the same manner as procedural statement `force` and `release`.
- If the `vpi_put_value` call contains a time delay information, VCS ignores it.
- If the `vpi_put_value` call contains a value with `vpiReleaseFlag`, VCS ignores the value argument of `vpi_put_value`.
- You cannot apply `vpi_put_value` to an entire struct. You can only apply it to individual members.

In the following example, SystemVerilog code makes a PLI call `$forceX`. In the PLI code, the SystemVerilog variables in that module are iterated and once the handle of the required variable is obtained, `vpi_put_value` is used with `vpiForceFlag` to force a value on to the variable. In the example, value 7 is forced on the variable through an argument (`value_s`) of `vpi_put_value` function call.

```

module dut(input int I, output int O);
int y;

always @(I) begin
    #10 y = I + 50;           // 2
    :
    #20 $forceX();          // 3
    :
    #20 $releaseX();        // 4
end
endmodule

// PLI code
:
void forceX()
{
    vpiHandle var;
    int flag = vpiForceFlag;

```

```

    s_vpi_value value_s = {vpiIntVal};
    value_s.value.integer = 7;
    :
    vpi_put_value(var, &value_s, NULL, flag);
    :
}

void releaseX()
{
    vpiHandle var;
    int flag = vpiReleaseFlag;
    s_vpi_value value_s = {vpiIntVal};
    value_s.value.integer = 70;
    :
    vpi_put_value(var, &value_s, NULL, flag);
    :
}

```

At time 40 ns, another PLI call \$releaseX is called, which will release the previously forced object.

With respect to the behavior, an object can be forced from within SystemVerilog and released from PLI or vice versa. Otherwise both force and release can happen from PLI. In all these cases the behavior of force and release would be the same as SystemVerilog force and release. All the rules that govern the force and release on various data types from within SystemVerilog code will also apply to VPI force and release.

SystemVerilog Operators

SystemVerilog includes the other assignment operators in C:

`=+ -= *= /= %= &= |= ^= <<= >>= <<<= >>>=`

The following table shows a few uses of these assignment operators and their equivalent in Verilog-2001.

operator example	Verilog-2001 equivalent
<code>b += 2;</code>	<code>b = b + 2;</code>
<code>b -= 2;</code>	<code>b = b - 2;</code>
<code>b *= 2;</code>	<code>b = b * 2;</code>

SystemVerilog includes the `++` increment and `--` decrement operators. The following table shows how they work.

operator example	Verilog-2001 equivalent
<code>++b; or b++;</code>	<code>b = b + 1;</code>
<code>--b; or b--;</code>	<code>b = b - 1;</code>

You can place the `++` and `--` operators to the left or the right of the variable operand. Doing so makes a difference when these operators and their operand are in a larger expression, but enabling you to do so is not yet implemented in VCS, so you can only use them as simple assignment statements.

New Procedural Statements

In SystemVerilog `if` and `case` statements, including `casex` and `casez`, can be qualified by the `unique` or `priority` keywords and there is a `do while` statement.

The unique and priority Keywords in if and case

Statements

The keyword `unique` preceding an `if` or nested `else if` statement specifies that one, and only one, conditional expression must be true. So in the following code:

```
unique if (l2!=0) $display("l2!=0");  
else if (l2==3) $display("l2=3");
```

There are two conditional expressions: `(l2!=0)` and `(l2==3)`.

VCS evaluates these conditional expressions in parallel and, if both are true, it is a warning condition and VCS displays the following warning message:

```
RT Warning: More than one conditions match in 'unique if'  
statement.  
    "filename.v", line line_number, at time      sim_time
```

If neither conditional expression is true, it is also a warning condition and VCS displays the following warning message:

```
RT Warning: No condition matches in 'unique if' statement.  
    "filename.v", line line_number, at time      sim_time
```

The keyword `priority` preceding an `if` or nested `else if` statement specifies that one conditional expression must be true. So in the following code:

```
priority if (l4!=0) $display("l4!=0");  
else if (l4==3) $display("l4=3");
```

There are two conditional expressions: `(l4!=0)` and `(l4==3)`.

VCS evaluates these conditional expressions in sequence to see if they are true. VCS executes the first statement controlled by the first conditional expression that is true. In this example, therefore, VCS would display:

```
l4!=0
```

The `priority` keyword allows more than one conditional expression to be true. In this example the conditional expression `(l4==3)` could also be true but this would not be a warning condition.

If neither conditional expression were true, it would be a warning condition and VCS displays the following warning message:

```
RT Warning: No condition matches in 'priority if' statement.  
            "filename.v", line line_number, at time sim_time
```

The keyword `unique` preceding a `case`, `caseX`, or `caseZ` statement specifies that one, and only one, case item expression can have the value of the case expression.

So in the following code:

```
unique case (l2)  
    0: $display("l2=%0d",l2);  
    1: $display("l2=%0d",l2);  
    !0: $display("l2 also !0");  
endcase
```

There is the case expression `l2` and three case item expressions: `0`, `1`, and `!0`.

VCS evaluates these case item expressions in parallel and, if more than one has the value of the case expression, it is a warning condition and VCS displays the following warning message:

RT Warning: More than one conditions match in 'unique case' statement.

```
"filename.v", line line_number, at time sim_time
```

If none of the case item expressions have the value of the case expression, it is also a warning condition and VCS displays the following warning message:

RT Warning: No condition matches in 'unique case' statement.

```
"filename.v", line line_number, at time sim_time
```

The keyword `priority` preceding a case, `case`, or `casez` statement specifies that one case item expression must have the value of the case expression. So in the following code:

```
priority case (l4)
  0: $display("l4 !1");
  1: $display("l4=%0d",l4);
endcase
```

If `l4`'s value is neither 0 or 1, it is a warning condition and VCS displays the following warning message:

RT Warning: No condition matches in 'priority case' statement.

```
"filename.v", line line_number, at time sim_time
```

The do while Statement

SystemVerilog has the `do while` statement. It's an alternative to the Verilog `while` statement, where an action is performed and then a condition is evaluated. Here is an example:

```
do #1 i1++; while (r1 == 0);
```

Here VCS repeatedly does these two things: increments signal i1 and then check to see if signal r1 is at 0. If when it checks r1 is no longer at 0, it stops incrementing i1. With a `while` statement, VCS would check on r1 before incrementing i1.

SystemVerilog Processes

SystemVerilog identifies the Verilog `always` block and its three variations, as static processes (There are dynamic processes that are not yet implemented). These three variations are as follows:

- `always_comb`
- `always_latch`
- `always_ff`

SystemVerilog also sees continuous assignment statements as static processes and you can use them to continuously assign to not just nets but also variables.

SystemVerilog also has a final block that executes during the last simulation time step.

The `always_comb` Block

An `always_comb` block models combinational logic. This block is to circumvent the problem of an `always` block with a missing `else` statement resulting in an unintended latch. The following is an example of an `always_comb` block:

```
always_comb
begin
```

```

    if (!mode)
        var1 = sig1 + sig2;
    else
        var1 = sig1 - sig2;
    var2 = sig1 + sig2;
end

```

SystemVerilog uses the term sensitivity list and in an `always_comb` block, the signals in the right-hand side of the assignment statements are inferred to be in the sensitivity list, meaning that any transition in these signals causes the execution of the `always_comb` block. In this example any transition in signals `sig1` and `sig2` cause the `always` block to execute.

To make sure that there is consistency between the expressions on the right-hand side of the assignments and the variables on the left-hand side, an `always_comb` block also executes at time zero, after the `initial` and `always` blocks that start at time zero begin executing.

There is a rule that there cannot be any other procedural assignment statements in the rest of the design that assign values to variables on the left-hand side of the assignment statements in an `always_comb` block. In this example there can be no other assignment statements in the design assigning values to `var1` and `var2`.

An `always_comb` block is similar to the `always` block with an implicit event control expression list. For example:

```

bit sig1 = 1;
bit sig2 = 1;

always @*
begin
    if (!mode)

```



```

    var1 = sig1 + sig2;
else
    var1 = sig1 - sig2;
    var2 = sig1 + sig2;
end

```

This `always` block executes at time zero only because there were transitions on the signals in its implicit sensitivity list at time zero, in this example on signals `sig1` and `sig2`. If there were to such time zero transitions, this `always` block would not execute at time zero. The `always_comb` block always executes at time zero so the `always_comb` block more accurately models proper wired logic behavior.

Another difference between such an `always` block and a similar `always_comb` block is that an `always_comb` block executes after all the `always` and `initial` blocks execute at time zero. The `always` block with an implicit event control expression list, if it executes at time zero, executes in no predictable order with the other `always` or `initial` blocks that execute at time zero.

If you have more than one `always_comb` block, there is no way to predict the order in which they execute at time zero other than that they execute after the `always` and `initial` blocks that execute at time zero.

You can consider an `always_comb` block. For example:

```

always_comb
bit1 = bit2 || bit3;

assign wire1 = bit1;

```

To be analogous to a continuous assignment statement:

```

assign wire1 = bit2 || bit3;

```

The `always_latch` Block

The `always_latch` block models latched behavior, combinational logic with storage. The following is an example of an `always_latch` block:

```
always_latch
if (clk) sigq <= sigd;
```

The `always_latch` block is similar to the `always_comb` block in the following ways:

- It has an inferred sensitivity list. It executes when there is a transition in the signals in the right-hand side of assignment statements.
- There can be no other assignment statements in the design that assign values to the variables in the left-hand side of its assignment statements.
- It always executes at time zero.

The difference between the `always_latch` and `always_comb` block is for synthesis. It's a way to make clear that you intend a latch for the code in the `always_latch` block.

The `always_ff` Block

The `always_ff` block models synthesizable sequential behavior. The following is an example of an `always_ff` block:

```
always_ff @(posedge clk or negedge reset)
    if (!reset)
        q <= 0;
    else
```

```
q <= d;
```

An `always_ff` block can only have one event control.

The final Block

The final block is a counterpart to the initial block. The final block executes during the last simulation time step in the simulation. A final block cannot contain delay specifications, non-blocking assignments, event controls, wait statements or contain user-defined task enabling statements for tasks that contain these constructs. The following is an example of a final block:

```
final
begin
mytask (l1,l2);
$display("          simulation ends at %0t", $time);
end
```

Final blocks are an LCA feature requiring a special license.

Tasks and Functions

SystemVerilog changes tasks and functions in the following ways:

- Easier ways to declare task and function ports
- Function inout and output ports
- Void functions
- Tasks no longer requiring `begin end` or `fork join` blocks

- Returning values from a task or function before executing all the statements in a task or function

Tasks

The following is an example of a valid SystemVerilog task. Note the differences from what would be a Verilog-2001 task.

```
task task1 (input [3:2][1:0]in1, in2, output bit
[3:2][1:0]out);
logic tlog1,tlog2;
tlog1=in1[3][1];
:
#10 out[3][1]=tlog1;
:
tlog2=in2;
endtask
```

Lets take a look at a number of lines in the task:

```
task task1 (input [3:2][1:0]in1, in2, output bit
[3:2][1:0]out);
```

The task header declares three ports:

- `in1` is declared as an `input` port. The keyword `input` is necessary only because it is a multi-dimensional packed array. In SystemVerilog a port can be a multi-dimensional array, packed or unpacked. In SystemVerilog task ports also have data types. The default data type is `logic`, so `in1` has the `logic` data type.

If `in1` were an unpacked multi-dimensional array, you would not need the keyword `input` to make it an `input` port:

```
in1 [3:2][1:0]
```

instead of

```
input [3:2][1:0] in1
```

- `in2` takes both defaults. The default direction is an `input` port and the default data type is `logic`.
- `out` deviates from both defaults and so it must be specified as an `output` port with the `bit` data type. It is also a multi-dimensional packed array.

```
logic tlog1,tlog2;
```

Local scalar variables `tlog1` and `tlog2` are declared, with the `logic` data type. Ports have a default data type, local variables do not.

```
tlog1=in1[3][1];  
:  
#10 out[3][1]=tlog1;  
:  
tlog2=in2;
```

Notice that these assignment statements are not inside a `begin end` or `fork join` block. By default statements in a task are executed sequentially, just like they were in a `begin end` block. If you want, you can enclose these statements in a `begin end` or `fork join` block.

Functions

The following is an example of a valid SystemVerilog function and the code that calls the function. Note the differences from what would be a Verilog-2001 function.

```

function reg [1:0] outgo(reg [3:2][1:0] in1,in2, output int
out);
int funcint;
funcint = in1[3] >> 1;
:
if (in2==0)
    return 0;
out = funcint;
:
outgo = funcint;
endfunction

initial
begin
:
#1 reg2=outgo(reg1,log1,int2);
:
end

```

Lets take a look at a number of lines in the function:

```

function reg [1:0] outgo(reg [3:2][1:0] in1,in2, output int
out);

```

The function header specifies that the function name is `outgo`. It declares that it returns a two-bit value of the `reg` data type (a SystemVerilog function can also return a multi-dimensional array, or a structure or union). The default data type of the return value is `logic`. The header declares three ports:

- `in1` is an input port of the `reg` data type. It is a multi-dimensional packed array. In SystemVerilog, function ports, like task ports, can be a multi-dimensional array. Also, like task ports, function ports default to the `input` direction, so port `in1` is an `input` port.

- `in2` is a scalar port that takes both defaults, `input` direction and the `logic` data type.
- `out` is an `output` port so the direction must be specified. It is of the `int` data type so that too is specified. In SystemVerilog a function can have an `output` or an `inout` port.

```
int funcint;
```

Local scalar variable `funcint` is declared, with the `int` data type. Ports have a default data type, local variables do not.

```
funcint = in1[3] >> 1;
:
out = funcint;
:
if (in2==0)
    return 0;
outgo = funcint;
```

Notice that, just like SystemVerilog tasks, these assignment statements are not inside a `begin end` or `fork join` block. By default, statements in a function are executed sequentially, just like they were in a `begin end` block. If you want, you can enclose these statements in a `begin end` or `fork join` block.

In these procedural statements:

1. The local variable `funcint` is assigned a shifted value of an element in the multi-dimensional array of input port `in1`.
2. The new value of `funcint` is assigned to the output port named `out`.
3. SystemVerilog functions can have a `return` statement that overrides an assignment to the function name. Here if input port `in2` equals zero, the function returns zero.

4. If the value of `in2` is not zero, the value of the local variable `funcint` is returned by the function.

```
#1 reg2=outgo(reg1,log1,int2);
```

In this statement that calls the function (SystemVerilog function calls are expressions unless they are `void` functions), signal `reg2` is assigned the return value of the function. Signals `reg1` and `log1` input values to the function and the value of the output port is a structural driver of signal `int2`.

SystemVerilog also allows `void` functions that do not have a return value. A `void` function is called as a statement not as an expression, as it is in non-void functions. The following is an example of a `void` function and the code that calls it:

```
function void display(bit in1);
bit funcbit;
funcbit=in1;
$display("bit1=%0b", funcbit);
//return 1'bz;
endfunction

initial
begin
bit1=1;
display(bit1);
end
```

A void function cannot contain a `return` statement.

Passing Arguments by Setting Defaults

You can specify default initial values for function and task ports. Default values can be constants or expressions, cross-module references, or class-references.

You can specify default values for:

- Input ports
- Inout and ref ports (one-value arguments only; constant arguments not supported)
- Automatic tasks/functions
- Task/functions contained in classes or methods of classes
- Dynamic/Associative arrays (available with `+svtb` switch only)
- `export` and `import` tasks and functions in interfaces

In the following example the function `func1` has two integer parameters whose default values are 100 and 300 respectively. This function returns an integer value. This function is defined in module `m` and is called in an initial block of module “`m`”;

```
module m;
    function int func1(int x = 100, int y = 300);
        return (x + y);
    endfunction

    initial
        func1 ();

endmodule
```

The function `func1` receives the default value of the arguments as if the function call were written:

```
func1(100, 300);
```

The same function `func1` can be called with a single argument:

```
initial
    func1(2);
```

In the first location, the argument overrides the default and the second default value is applied as if the function call were written as:

```
func1(2, 300);
```

SystemVerilog Packages

A package is a scope that contains declarations of things that you want to share with more than one module, macromodule, interface or program.

Note:

- A SystemVerilog package is a concept borrowed from VHDL.
- Classes in packages are an LCA feature requiring a special license.
- SystemVerilog assertion sequence and property declaration in packages, and referencing them in a module or program definition, is not supported yet.

The things that you can declare in a package are data types, including complex data types like structures, and user-defined tasks and functions.

The following are examples of a package definitions and how to reference things declared in a package:

```
package pack1;  
int int1;  
endpackage
```

You define a SystemVerilog package between the `package` and `endpackage` keywords. You must specify an identifier for the package. This package declares `int` variable `int1`.

A module, macromodule, interface, or program references something declared in a package with an `import` statement with the `::` scope resolution operator.

You can also use an `import` statement, with the `::` scope resolution operator, in a package, to reference the contents of another package, but you can't use it to reference something in a module, macromodule, interface, or program. The following is an example of a package referencing something in another package:

```
package pack2;
import pack1::int1;
endpackage
```

What follows are two module definitions that share what is declared in package `pack1`:

```
module top1;
import pack1::int1;

initial
begin
$monitor ("int1=%0d at %0t", int1, $time);
#10 int1=11;
#10 $finish;
end

endmodule
```

```
module top2;
import pack1::int1;

initial
```

```
#6 int1=7;

endmodule
```

Modules `top1` and `top2` share `int1` in package `pack1`. They both declare the use of the variable with `import` statements that reference both the package and the variable. The `$monitor` system task displays the following:

```
int1=0 at 0
int1=7 at 6
int1=11 at 10
```

Both modules `top1` and `top2` assign values to `int1`.

Note:

A data type declaration in a package is different from a data type declaration in `$root`. you must reference a data type in a package with an `import` statement before you can assign values to it.

The following package declares a user-defined data type.

```
package pack1;
typedef enum { FALSE, TRUE } bool_values;
endpackage
```

The following module definition references the user-defined data type and the values of the user defined data type using `import` statements:

```
module top;
import pack1::bool_values;
import pack1::FALSE;
import pack1::TRUE;
bool_values top_bool_value1;
bool_values top_bool_value2 = TRUE;
```

```

initial begin
    #1 top_bool_value1 = top_bool_value2;
    #5 top_bool_value2 = FALSE;
end

initial
$monitor("%0t top_bool_value1=%0d top_bool_value2=%0d",
$time, top_bool_value1, top_bool_value2);

endmodule

```

The `$monitor` system task displays the following

```

0 top_bool_value1=0 top_bool_value2=1
1 top_bool_value1=1 top_bool_value2=1
6 top_bool_value1=1 top_bool_value2=0

```

The following package contains a structure and a user-defined function:

```

package pack1;

typedef struct {real real1, real2;} struct1;

function struct1 halvfunc1 (struct1 in1);
    halvfunc1.real1 = in1.real1 / 2;
    halvfunc1.real2 = in1.real2 / 2;
endfunction

endpackage : pack1

```

The following module definition begins with the following:

1. A wildcard character in the `import` statement to specify referencing both the structure and the user-defined function in the module
2. Four declarations of instances of the structure

```

module mod1;
import pack1::*;
struct1 mod1struct1;
struct1 mod1struct2;
struct1 mod1struct3;
struct1 mod1struct4;

initial
begin
mod1struct1.real1=5;
mod1struct1.real2=11;
mod1struct2.real1=3;
mod1struct2.real2=7;
#10 mod1struct3 = halvfunc1 (mod1struct1);
#10 mod1struct4 = halvfunc1 (mod1struct2);
#10 $display("mod1struct3.real1=%0f",mod1struct3.real1);
#10 $display("mod1struct3.real2=%0f",mod1struct3.real2);
#10 $display("mod1struct4.real1=%0f",mod1struct4.real1);
#10 $display("mod1struct4.real2=%0f",mod1struct4.real2);
end

endmodule

```

The `$display` system tasks display the following:

```

mod1struct3.real1=2.500000
mod1struct3.real2=5.500000
mod1struct4.real1=1.500000
mod1struct4.real2=3.500000

```

Exporting Time Consuming User-Defined Tasks with the SystemVerilog DPI

You can export a user-defined task that contains delays into the C or C++ language using the DPI. Such a user-defined task consumes simulation time and does not start and finish its execution during the same simulation time step.

Exporting time-consuming user-defined tasks is an LCA feature requiring a special license.

Time-consuming user-defined tasks are also called blocking tasks, they suspend, for some simulation time, the C or C++ function that calls it.

The following is an example of a module definition containing such a user-defined task:

```
`timescale 1 ns/1 ns
module test;

    import "DPI" context task func_in_C(int i);

    task task_in_SV(inout int i);
        #5 i = i/2;
    endtask

    export "DPI" task task_in_SV;

initial
    func_in_C(4);

endmodule
```

One of the first lines in the module declares the use of a C language function later in the code:

```
import "DPI" context task func_in_C(int i);
```

This line says that we are importing (calling) a function in the C or C++ language using the DPI. There is the following required information in this declaration:

- The `context` keyword enables, in this case, the C or C++ language function to call the user-defined task in the SystemVerilog code. (This keyword also has other uses in the DPI.)
- The `task` keyword also enables the C or C++ language function to call the user-defined task in the SystemVerilog code. We are calling it a task even though there are no tasks in C or C++.

You must include both keywords.

The SystemVerilog IEEE Std 1800-2005, Section 26.1.1 "Tasks and functions", specifies the following:

“All functions used in DPI are assumed to complete their execution instantly and consume 0 (zero) simulation time, just as normal SystemVerilog functions.”

So imported C functions that call time-consuming user-defined tasks must be declared to be tasks.

The SystemVerilog IEEE Std 1800-2005, Section 26.7 "Exported tasks", specifies the following:

- “It is never legal to call an exported task from within an imported function.”
- “It is legal for an imported task to call an exported task only if the imported task is declared with the `context` property.”

Next comes the user-defined task:

```
task task_in_SV(inout int i);
    #5 i = i/2;
endtask
```


Notice that there is a delay in this user-defined task. This is a blocking task for the C or C++ function that calls it. The argument for this user-defined task has the `inout` direction.

Next comes the declaration that the user-defined task can be called by a C or C++ function:

```
export "DPI" task task_in_SV;
```

The module ends with procedural code that calls the C or C++ function:

```
initial  
    func_in_C(4);
```

The C or C++ source code is as follows:

```
#include <svdpi.h>  
  
extern int task_in_SV(int *i);  
  
void func_in_C(int i)  
{  
    printf("before export, i=%d\n",i);  
    task_in_SV(&i);  
    printf("after export, i=%d\n",i);  
}
```

The `#include` preprocessor statement:

```
#include <svdpi.h>
```

Specifies declarations and prototypes of SystemVerilog DPI library functions and tasks. Synopsys recommends including this line before calls to DPI functions and tasks.

The `extern` declaration declares the user-defined function.

The address of the `int` variable `i` is passed to the user-defined task because the argument of the task has the `inout` direction.

The `printf` statements display the following:

```
before export, i=4  
after export, i=2
```

Hierarchy

SystemVerilog contains enhancements for representing the design hierarchy:

- The `$root` top-level global declaration space
- New data types for ports
- Instantiation using implicit `.name` connections
- Instantiation using implicit `.*` connections
- New port connection rules for variables

The `$root` Top-Level Global Declaration Space

In SystemVerilog there is the `$root` top-level declaration space which not only contains all uninstantiated modules, but also interface definitions (in the current implementation interface definitions cannot be inside module definitions or other interfaces), user-defined tasks and functions, parameter, nets and variables, and type definitions.

Some examples of `$root` declarations are as follows:

```

parameter msb=7;

typedef int myint;

wire w1,w2,w3,w4;

logic clk;

and and1 (w2,w3,w4);

tran tr1 (w1,w4);

primitive prim1 (out,in);
input in;
output out;
table
// in : out
    0 : x;
    1 : 0;
    x : 1;
endtable
endprimitive

event recieved_data;

task task1 (input [3:2][1:0]in1, in2, output bit
[3:2][1:0]out);
logic tlog1,tlog2;
tlog1=in1[3][1];

:
#10 out[3][1]=tlog1;

:
tlog2=in2;
endtask

function void left (output myint k);
    k = 34;
    $display ("entering left");
endfunction

interface try_i;

```

```

wire [7:0] send, receive;
endinterface

module top1(w1);
output w1;
endmodule

module top2(w1);
input w1;
endmodule

```

All constructs that are declared or defined in `$root` are, for the most part, accessible to the entire design. Any module, regardless of its place in the hierarchy, can use the parameter, use the type, read or write to these variables, use the named event, call the task and function, or instantiate the interface. The gate and switch primitive cannot be instantiated in the rest of the design (they are already instantiated in `$root`) but the rest of the design can write to their inputs and read their outputs. The UDP can be instantiated in the rest of the design. The module definitions, not instantiated elsewhere, are top-level modules. Note that they connect to `$root` level wire `w1`.

New Data Types for Ports

In SystemVerilog a module `input` or `output` port can be any net data type or any variable data type including an array, a structure, or a union. For example:

```

typedef struct {
    bit bit1;
    union packed{
        int int1;
        logic log1;
    } myunion;
} mystruct;

```

```
module mod1(input int in1, byte in2, inout wire io1, io2,  
           output mystruct out);  
  
:  
endmodule
```

In the module header for module mod1:

1. The first port is named in1. It is specified as an `input` port with the `int` data type.
 - If we omitted both the direction and data type, SystemVerilog expects the port to be declared following the header.
 - If only the direction is omitted, it defaults to `inout`.
 - If only the data type is omitted, it defaults to the `wire net` data type (which you can change with the ``default_nettype` compiler directive to another net data type).
2. The second port is named in2. No direction is specified so it inherits the direction from the previous port, so in2 is an `input` port with the `byte` data type.
3. The third port is named io1. It's specified as an `inout` port with the `wire` data type.
4. The fourth port is named io2. Not being the first port in the list, it inherits the direction and data type from port io1.
5. The last port is named out. It is an output port that is the structure `mystruct`.

You still can only use net data types for `inout` ports.

The Accellera SystemVerilog 3.1a specification says that named events can also be ports, but this is not implemented in VCS.

Instantiation Using Implicit .name Connections

In SystemVerilog if the name and size of a port matches the name and size of the signal that connects to that port, you can make connections in any order without matching the order of the ports or using a name based connection list where you have to enter each port and the signal that connects to it. For example:

```
module top;
logic [7:0] log1;
byte byt1 [1:0];

dev dev1(.log1,.byt1);
endmodule

module dev(input byte byt1 [1:0], input logic [7:0] log1);
:
endmodule
```

Module top instantiates module dev. In the module instantiation statement in module top, the connection list has signal log1 first, followed by signal byt1. In the module header for module dev, the port connection list has port byt1 first followed by port log1.

In Verilog-2001 or Verilog-1995 you would need a name-based connection list in the module instantiation statement:

```
dev dev1(.log1(log1),.byt1(byt1));
```

Instantiation Using Implicit .* Connections

In SystemVerilog, if the name and size of a port matches the name and size of the signal that connects to that port, you use a period and

as asterisk to connect the signals to the ports, similar to an implicit event control expression list for an always block. For example:

```
module top;
logic [7:0] log1;
byte byt1 [1:0];

dev dev1(.*);
endmodule

module dev(input byte byt1 [1:0], input logic [7:0] log1);
:
endmodule
```

New Port Connection Rules for Variables

SystemVerilog allows you to declare an `input` port to be a variable. This is another way to use a structural driver, see ["Writing To Variables" on page 22-19](#). If you declare an `input` port to be a variable, you cannot have multiple drivers, so you cannot do any of the following:

- Assign values to the variable with a procedural assignment statement or a user-defined task enabling statement.
- Assign values to it with a continuous assignment statement or have values propagate to it from a module instance, gate, switch-level primitive, or UDP.

Like Verilog-2001 and Verilog-1995, SystemVerilog allows you to declare an `output` port to be a variable and prohibits the same for `inout` ports.

Ref Ports on Modules

Like arguments to tasks that you declare with the `ref` keyword, you can declare module ports with the `ref` keyword instead of the `input`, `output`, or `inout` keywords.

A `ref` port is a reference to the signal that connects to it in a module instantiation statement in a higher level module instance. This connected higher-level signal is called a highconn signal. For a `ref` port there isn't separate simulation values for the highconn signal and the instance's port, with values propagating to the port from the signal or to the signal from the port. VCS only has one copy of the simulation data for both the highconn signal and the `ref` port. All operations that the module does to its `ref` port it also does directly to the highconn signal.

The following is an example of the use of a `ref` port:

```
`timescale 1 ns / 1 ns
module refportmod (ref integer refin1);
always @ refin1
#1 refin1 = refin1/2;
endmodule

module test;
integer int1;

initial
begin
$monitor("int1 = %0d at %0t",int1,$time);
#10 int1 = 100;
#10 int1 = 66;
#10 int1 = 24;
end

refportmod refportmod1 (int1);
```



```
endmodule
```

In module `refportmod` the port named `refin1` is declared with the `ref` keyword. All operations done in module `refportmod` to `ref` port `refin1` also happen to the value of the `highconn` signal, `int1`, in module `test`.

The `$monitor` system task in module `test` displays the following:

```
int1 = x at 0  
int1 = 100 at 10  
int1 = 50 at 11  
int1 = 66 at 20  
int1 = 33 at 21  
int1 = 24 at 30  
int1 = 12 at 31
```

The value of integer `int1` is halved because it is connected to a `ref` port in a module that halves this port.

A `ref` port is the only way to share a variable value across the hierarchy.

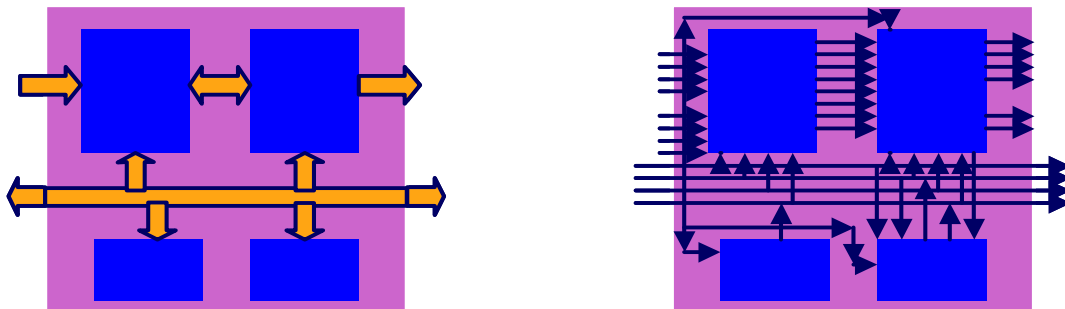
Both the `ref` port and the `highconn` signal must have the same variable data type.

A `ref` port differs from an `inout` port in that an `inout` port must have a net data type, whereas a `ref` port must have a variable data type.

Interfaces

Interfaces were developed because most bugs occur between blocks in a design and interfaces help eliminate these wiring errors. Interfaces are a way of encapsulating the communication and interconnect between these blocks, but they are more than just that. They help you to develop a divide and conquer methodology and are reusable in other places in a design and in other designs.

Figure 22-4 Block Diagrams



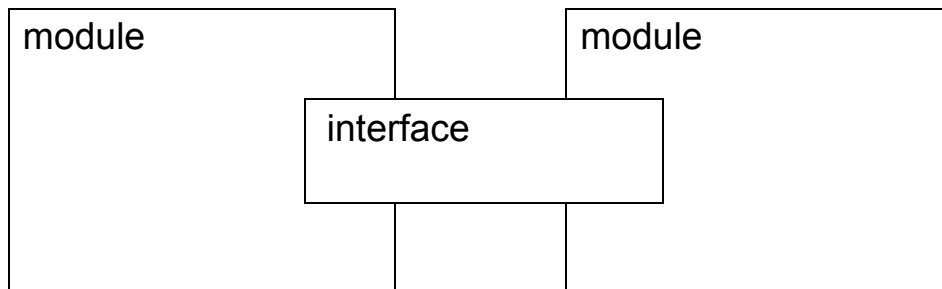
Consider the wide arrows in the block diagram on the left to be interfaces. They are more than just wire bundles, They are an implementation of how to communicate between blocks. As interfaces, they help you to focus on how information is communicated between block.

At its simplest level, an interface encapsulated communication like a struct encapsulates data:

```
typedef struct{                interface intf;
int int1;                      int int1;
logic [7:0] log1;             wire [7:0] w1;
} s_type;                      endinterface
```

Think of a wire as a built-in interface.

An interface is like a module that straddles other modules.



Interfaces help you to maintain your code. For example, to add a signal between blocks in your design, you only need to edit the interface instead of editing the module headers of both modules and the module instantiation statements for these modules. Interfaces are not just wires. Interfaces can contain the following:

- Variables and parameters that can be declared in one location and used in various modules.
- Tasks or functions that can be used by all instances that connect to these interfaces.
- Procedures for checking and other verification operations.

Example 22-5 introduces a basic interface and the code that uses it.

Example 22-5 Basic Interface

```
interface intf;
wire [7:0] send, receive;
endinterface

module test;
logic [7:0] data_in1, data_in2, data_out1, data_out2;

intf intf1();

sendmod sm1 (intf1, data_in1, data_out1);
receivemod rm1 (intf1, data_in2, data_out2);

endmodule

module sendmod (intf intf1,
               input logic [7:0] in,
               output logic [7:0] out);

assign out = intf1.receive;
assign intf1.send = in;
endmodule

module receivemod(intf intf1,
                 input logic [7:0] in,
                 output logic [7:0] out);

assign out = intf1.send;
assign intf1.receive = in;
endmodule
```

Interface defined in \$root

Interface instantiated in module definition

Connect module instance to interface instance

Interface declared in module header

Reading from a signal in an interface

Writing to a signal in an interface

As illustrated in this example:

- In the VCS implementation, interface definitions must be in the `$root` declaration space, outside of any module definition. In the Accellera SystemVerilog 3.1a specification, this is not the case,

and interface definitions can be in module definitions or nested in other interface definitions.

- To use an interface to connect a module instance to another, you must instantiate the interface in the module that instantiates the two module instances.
- You also must declare the interface in the port connection list of the module headers of the two modules you are connecting with the interface. Note that the interface instance names in the port connection lists do not have to match the interface instance name in the module that instantiates these module, not do the instance names have to match in the port connection lists of the modules connected by the instance. In this example we have the following instance names for the same interface: `intf1`, `intfa1`, and `intfb1`.
- To read from or write to a signal in an interface, you reference it by *`interface_instance_name.signal_name`*.

This basic example, meant only to introduce interfaces, doesn't do much to show you the utility of an interface. In fact, in this example the signals in the interface, `send` and `receive`, are functionally the same as `inout` ports in the module definitions and two nets with the `wire` data type connecting these `inout` ports.

What if your intent is for the instance of module `sendmod` to always send data to module `receivemod` through one signal, in this case signal `send`, and receive data from module `receivemod` from the other signal, signal `receive`? This basic interface isn't doing the job for you. You can use an interface construct called a `modport` to add directionality to signals in an interface.

Using Modports

A modport specifies direction for a signal from a “point of view.” With these two signals in the interface we can specify two modports or “points of view” and the two modules connected by the interface will use different modports, and have different points of view.

Let’s modify the interface definition by adding two modports.


```
interface intf;
logic [7:0] send, receive;
modport sendmode (output send, input receive);
modport receivemode (input send, output receive);
endinterface
```

Modules that use `modport sendmode`, have an `output` port named `send` and an `input` port named `receive`. Modules that use `modport receivemode`, have an `input` port named `send` and an `output` port named `receive`.

The data type is also changed. This isn’t done to enable the modport. It’s to enable the modules connected by this interface, that we will also modify, to make procedural assignments to the signals.

Now let's modify the module definition for module sendmod:

modport follows the
interface name



```
module sendmod (intf.sendmode intf1,  
               input logic [7:0] in,  
               output logic [7:0] out);  
always @(intf1.receive) out = intf1.receive;  
always @(intf1.receive) intf1.send = in;  
  
endmodule
```

In the module header, in the connection list in the header where using the interface is declared, the `modport` is also declared, using the following syntax:

```
interface_name.modport_name
```

Module receivemod is also modified:

```
module receivemod(intf.receivemode intf1,  
                 input logic [7:0] in,  
                 output logic [7:0] out);  
always @(intf1.send) out = intf1.send;  
always @(intf1.send) intf1.receive = in;  
endmodule
```

Modports also control the visibility of signals declared in an interface. If a signal in an interface is not specified in a `modport`, then modules that use the `modport` cannot access the signal.

Functions In Interfaces

If we define a user-defined task or function in the interface, we can use the `import` keyword in the `modport` to make it accessible to the module instances connected by the interface and that use the `modport`. For example:

```
interface intf;
  logic [7:0] send, receive;

  function automatic logic parity(logic [7:0] data);
  return (^data);
endfunction

modport sendmode (output send, input receive,
                  import function parity());
modport receivemode (input send, output receive,
                    import function parity());
endinterface
```

Using the keyword `import` and specifying whether it is a task or function in the `modport`, enables modules connected by the interface to use the function named `parity`. Using a function in an interface is called using a method.

```
module sendmod (intf.sendmode intf1,
                input logic [7:0] in,
                output logic [7:0] out,
                output logic out_parity);

  always @(intf1.receive)
  begin
    out = intf1.receive;
    out_parity = intf1.parity(intf1.receive);
    intf1.send = in;
  end

endmodule
```


This module uses the method called parity, using the syntax:

```
interface_instance_name.method_name
```

Enabling SystemVerilog

You tell VCS to compile and simulate SystemVerilog code with the `-sverilog` compile-time option. No runtime option is necessary.

IMPORTANT:

Radiant Technology (+rad) 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.

Disabling unique And priority Warning Messages

By default VCS displays warning messages in certain situations when you enter `unique if`, `unique case`, `priority if` and `priority case` statements. For example:

```
RT Warning: More than one conditions match in 'unique if' statement.
```

```
    "filename.v", line line_number, at time      sim_time
```

```
RT Warning: No condition matches in 'unique if' statement.
```

```
    "filename.v", line line_number, at time      sim_time
```

RT Warning: No condition matches in 'priority if' statement.
"filename.v", line *line_number*, at time *sim_time*

RT Warning: More than one conditions match in 'unique case' statement.
"filename.v", line *line_number*, at time *sim_time*

RT Warning: No condition matches in 'unique case' statement.
"filename.v", line *line_number*, at time *sim_time*

RT Warning: No condition matches in 'priority case' statement.
"filename.v", line *line_number*, at time *sim_time*

You can suppress these warning messages with the following compile-time option and keyword argument.

`-ignore keyword_argument`

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.

23

SystemVerilog Assertion Constructs

SystemVerilog assertions (SVA), just like OpenVera assertions (OVA), are a shorthand way to specify how you expect a design to behave and have VCS display messages when the design does not behave as specified. You can use both to accomplish the same thing. SystemVerilog assertions are in the SystemVerilog 3.1a standard promulgated by Accellera, the electronics industry wide organization for the advancement of hardware description languages. OpenVera assertions are part of the Synopsys proprietary OpenVera standard.

VCS has implemented both types of SystemVerilog assertions:

- Immediate assertions
- Concurrent assertions

Immediate assertions are a test of an expression when VCS executes the immediate assertion. An immediate assertion is a statement in procedural code.

Concurrent assertions specify how the design behaves during a span of simulation time.

Immediate Assertions

An immediate assertion resembles a conditional statement in that it has a boolean expression that is a condition. When VCS executes the immediate assertion it tests this condition and if it is true, VCS executes some statements in what is called a pass action block. If the condition is not true VCS executes statements in what is called a fail action block.

The following is an example of an immediate assertion:

```
module test;
  :
  initial
  begin:named
  :
  a1:assert (lg1 && lg2 && lg3)
    begin: pass
      $display("%m passed");
    :
  end
else
  begin: fail
    $display("%m failed");
  :
  end
end
```

In this example the immediate assertion is labeled a1, and its expression (lg1 && lg2 && lg3) is the condition. If the condition is true, VCS executes the begin-end block labeled pass. This is the pass action block (it is not required to name this action block). If the condition is not true, VCS executes the begin-end block labeled fail (it is also not required to name this action block), it follows the keyword else.

If, for example, this immediate assertion passes, the condition is true, and VCS displays the following:

```
test.named.a1 passed
```

Concurrent Assertions Overview

Concurrent assertions consists of one or more properties. A property consists of a clock signal and one or more sequences. In a property you can either specify a sequence of values on signals and the simulation time that occurs between these values, specified as clock ticks, or instantiate a sequence that you declare. You can declare a sequence and then use it as a building block in a property.

Sequences

A sequence enables you to build and manipulate sequential behavior. The following is an example of a sequence:

```
sequence s1;  
sig1 ##1 sig2;  
endsequence
```

Sequence s1 specifies that signal sig1 is true and then one clock tick later signal s2 is true. In this case the clock signal and the edge that specifies the clock tick, are in a property definition that instantiates this sequence. The ## operator specifies a delay of a number of clock ticks (the value you specify must be a non-negative integer).

You can specify any number of signals in the sequential expression and the logical negation operator. For example:

```
sequence s2;  
sig1 ##1 sig2 ##2 !sig3;  
endsequence
```

Sequence s2 specifies that signal sig1 must be true, and one clock tick later signal s2 must be true, and then two clock ticks after that signal s3 must be false.

You can use a sequence in another sequence as a building block in another sequence, for example:

```
sequence s1;  
sig1 ##1 sig2;  
endsequence
```

```
sequence s3;  
s1 ##2 !sig3;  
endsequence
```

Here sequence s1 is used in sequence s3.

You can declare a sequence in the following places in your code:

- In a module definition
- In an Interface definition

- At `$root` (in SystemVerilog `$root` means outside of any other definition, a sequence defined in `$root` is globally accessible).

Note:

The SystemVerilog LRM says that you can declare a sequence in a module definition but never in an `always` or `initial` block.

Using Formal Arguments In A Sequence

You can specify formal arguments in a sequence and then make substitutions when you use it in another sequence. For example:

```
sequence s1(sig3,sig4);
sig3 ##1 sig4;
endsequence
```

```
sequence s2;
s1(sig1,sig2) ##1 sig5;
endsequence
```

Specifying a Range of Clock Ticks

You can specify a range of clock ticks in the delay. For example:

```
sequence s1;
sig1 ##[1:3] sig2;
endsequence
```

This sequence specifies that `sig1` must be true and then `sig2` must be true at either the first, second, or third subsequent clock tick.

You can specify that the range end at the end of the simulation with the `$` token, for example:

```
sequence s1;
sig1 ##[1:$] sig2;
endsequence
```

The operands in the sequences need to be boolean expressions, they do not have to be just signal names. For example:

```
sequence s1;  
sig1 == 1 ##1 sig3 || sig4;  
endsequence
```

Unconditionally Extending a Sequence

You can unconditionally extend a sequence by using the literal true value 1 or a text macro defined as 1. For example:

```
sequence s2;  
sig1 ##1 sig2 ##2 !sig3 ##3 1;  
endsequence
```

Here sig1 must be true, one clock tick later sig2 must be true, then two clock ticks later sig3 must be false, but the sequence doesn't end until three clock ticks later. Extending a sequence can be handy when you are using a sequence in another sequence.

Using Repetition

There are three operators for specifying the repetition of a sequence:

- The consecutive repetition operator [$*$
- The goto repetition operator [$->$
- The non-consecutive repetition operator [$=$

The consecutive repetition operator [$*$ is for specifying consecutive repetitions of a sequence. For example, the following sequence:

```
sequence s1;
```



```
sig1 ##1 sig2 ##1 sig2 ##1 sig2;  
endsequence
```

Can be shortened to the following:

```
sequence s1;  
sig1 ##1 sig2 [*3];  
endsequence
```

Note:

The value you specify with the [* operator must be a positive integer.

You can use repetition in a range of clock ticks. For example:

```
sequence s1;  
(sig1 ##2 sig2) [*1:3];  
endsequence
```

This sequence specifies that the sequence is run at the length of itself, or the length of itself doubled, or tripled. This sequence is the equivalent of all of the following:

```
sequence s1;  
(sig1 ##2 sig2);  
endsequence
```

```
sequence s1;  
(sig1 ##2 sig2 ##1 sig1 ##2 sig2);  
endsequence
```

```
sequence s1;  
(sig1 ##2 sig2 ##1 sig1 ##2 sig2 ##1 sig1 ##2 sig2);  
endsequence
```

You can specify an infinite number of repetitions with the \$ token. For example:

```
sequence s1;
(sig1 ##2 sig2) [*1:$];
endsequence
```

Note:

##1 is automatically added between repetitions.

The goto repetition operator [-> (non-consecutive exact repetition) specifies the repetition of a boolean expression, such as:

```
a ##1 b [->min:max] ##1 c
```

This is equivalent to:

```
a ##1 ((!b [*0:$] ##1 b)) [*min:max] ##1 c
```

Adding the range specification to this allows the construction of useful sequences containing a boolean expression that is true for at most N occurrences:

```
a ##1 b[->1:N] ##1 c // a followed by at most N occurrences
                      // of b, followed by c
```

The non-consecutive repetition operator [= extends the goto repetition by extra clock ticks where the boolean expression is not true.

```
a ##1 b [=min:max] ##1 c
```

This is equivalent to:

```
a ##1 ((!b [*0:$] ##1 b)) [*min:max] ##1 !b[*0:$] ##1 c
```

The above expression would pass the following sequence, assuming that 3 is within the min:max range.

```
a c c c c b c c b c b d d d c
```

Specifying a Clock

You can specify a clock in a sequence. For example:

```
sequence s1;  
@(posedge clk) sig1 ##1 sig2;  
endsequence
```

This sequence specifies that the clock tick is on the rising edge of signal clk.

Value Change Functions

You can also include the following system functions in a sequential expression. They tell you about value changes between clock ticks:

`$sampled(expression)`

Returns the sampled value of the expression with respect to the last occurrence of the clocking event.

`$rose(expression)`

If the expression is just a signal, this returns 1 if the least significant bit of the signal changed to 1 between clock ticks. If the expression is more than one signal and an operator, for example `sig1 + sig2`, this returns 1 if the value of the least significant bit in the evaluation of the expression changes from 0, X, or Z to 1.

`$fell (expression)`

If the expression is just a signal, this returns 1 if the least significant bit of the signal changed to 0 between clock ticks. If the expression is more than one signal and an operator, for example `sig1 + sig2`, this returns 1 if the value of the least significant bit in the evaluation of the expression changes from 1, X, or Z to 0.

`$stable (expression)`

Returns 1 if the value of the expression does not change between clock ticks. A change in a four state signal from X to Z returns false.

The following is an example of using these system functions:

```
sequence s1;
$rose(sig1) ##1 $stable(sig2 && sig3);
endsequence
```

Anding Sequences

You can use the `and` operator to specify that two sequences must occur (succeed), but not necessarily at the same time. The following is an example:

```
sequence s1;
sig1 ##1 sig2;
endsequence
```

```
sequence s2;
sig3 ##2 sig4;
endsequence
sequence s3;
s1 and s2;
endsequence
```

Sequence `s3` succeeds when both sequences `s1` and `s2` succeed. The time of the success of `s3` is whenever the last of `s1` or `s2` succeed.

Intersecting Sequences (And With Length Restriction)

You use `intersect` operator to specify the match of the operand sequential expressions at the same clock tick. For example:

```
sequence s1;
l1 ##1 l3;
endsequence

sequence s2;
l2 ##1 l4;
endsequence

sequence s3;
s1 intersect s2;
endsequence
```

In this example, sequence s3 can match because sequences s1 and s2 both have the same number of clock ticks, sequence s3 does match when sequences s1 and s2 match.

Oring Sequences

You can use the `or` operator to specify that one of two sequences must succeed. The following is an example:

```
sequence s1;
sig1 ##1 sig2;
endsequence

sequence s2;
sig3 ##2 sig4;
endsequence

sequence s3;
s1 or s2;
endsequence
```

Sequence `s3` succeeds when either sequences `s1` and `s2` succeed but only when sequences `s1` and `s2` start at the same time.

Only Looking For the First Match Of a Sequence

The `first_match` operator specifies that a sequential expression matches only once. After its first success, VCS no longer looks for subsequent matches.

```
sequence s1;  
first_match(l1 ##[1:2] l2);  
endsequence
```

In `s1`, if `l1` is true at the first clock tick, the expression could match at the next clock tick, or the one after that, but the use of the `first_match` operator means that VCS does not monitor for the second possible match if the first possible match occurs.

Conditions for Sequences

You can use the `throughout` operator to specify a condition that must be met throughout the sequence in order for the sequence to succeed. For example:

```
sequence s1;  
(sig3 || sig4) throughout sig1 ##1 sig2;  
endsequence
```

For this sequence to succeed, not only must `sig1` be true and then in the next clock tick, `sig2` be true, but also for both clock ticks the expression `(sig3 || sig4)` must evaluate to true.

Specifying That Sequence Match Within Another Sequence

You use the `within` operator to require that one sequence begin and match when or after another starts but before or when the other one matches. For example:

```
sequence s1;
l1 ##3 l4;
endsequence

sequence s2;
l2 ##1 l3;
endsequence

sequence s3;
s2 within s1;
endsequence
```

Sequence `s1` requires three clock ticks, sequence `s2` only requires one clock tick. So it is possible for `s2` to begin and end during `s1`, signal `l2` to toggle to true after `l1` does, a clock tick later, `l3` toggles to true, and `l4` toggling to true a clock tick later.

Using the End Point of a Sequence

Sequences have an `ended` method that you can use in another sequence to specify that the other sequence includes the end of the first sequence. For example:

```
sequence s1;
@(posedge clk) sig1 ##1 sig2;
endsequence

sequence s2;
s1.ended ##1 sig3;
endsequence
```

Note:

If you are referencing a sequence in another sequence, and you are using the ended method in the second sequence, the referenced sequence must specify its clock tick. The first sequence above does so by beginning with `@(posedge clk)`.

You cannot use the ended method on a sequence with formal arguments. Use (or instantiate) such a sequence in another sequence, and then you can use the method on the other sequence. For example:

```
sequence s1(sig3,sig4);
@(posedge clk) sig3 ##1 sig4;
endsequence
```

```
sequence s2;
s1(sig3,sig4);
endsequence
```

```
sequence s3;
s2.ended ##1 sig1;
endsequence
```

Level Sensitive Sequence Controls

You can use a SystemVerilog assertion sequence as an event control expression. You can also use a sequence as the conditional expression in a `wait`, `if`, `case`, `do while`, or `while` statements, if you use the `triggered` sequence method.

The `triggered` sequence method evaluates to true if the sequence successfully completes during the same time step. You use this method in an expression that includes the sequence name, immediately followed by a period (.) and then the keyword `triggered`.

For example:

```
if (sequence1.triggered)
```

Level sensitive sequence controls are documented in section 8.11, starting on page 93, of the *SystemVerilog 3.1a Language Reference Manual*.

The following annotated code example shows using a sequence for these purposes:

```
module test;
logic l1,l2,clk;

sequence s1;
@ (posedge clk) l1 ##1 l2;
endsequence
```

Sequence s1 specifies that when there is a rising edge on variable clk, variable l1 is true, and with the next rising edge on clk, variable l2 is true.

```
initial
begin
clk=0;
#4 l1=1;
#10 l2=1;
#3 $finish;
end

always
#5 clk=~clk;
```

There will be a rising edge on variable clk at time 5 and 15. Simulation ends at time 17. At the first rising edge, l1 will be true, at the second rising edge, l2 will be true. The sequence will occur.

```
always @(s1)
$display("sequence s1 event control at %0t\n", $time);
```

Sequence s1 is an event control expression.

```
initial
begin
wait (s1.triggered)
$display("wait condition s1.triggered\n");
```

The `triggered` method with sequence s1 is the conditional expression for the `wait` statement.

```
if (s1.triggered)
$display("if condition s1.triggered\n");
case (s1.triggered)
1'b1 : $display("case condition s1.triggered happened\n");
1'b0 : $display("s1.triggered did not happen\n");
endcase
do
begin
$display("do while condition s1.triggered\n");
$finish;
end
while (s1.triggered);
end

endmodule
```

The `triggered` method with sequence s1 is also the conditional expression for the `if`, `case`, and `do while` statements.

Sequence s1 does occur, so VCS displays the following:

```
sequence s1 event control at 15

wait condition s1.triggered

if condition s1.triggered
```

```
case condition s1.triggered happened
do while condition s1.triggered
```

Properties

A property says something about the design, so a property evaluates to true or false.

Concurrent assertions use properties and properties contain sequences, either instantiated or containing sequential expressions like a sequence. Both of the following sequences and all but the last of the following properties are valid:

```
sequence s1;
sig1 ##1 sig2;
endsequence
```

```
sequence s2;
@(posedge clk) sig3 ##1 sig4;
endsequence
```

```
property p1;
@(posedge clk) s1;
endproperty
```

```
property p2;
@(posedge clk) s2;
endproperty
```

```
property p3;
@(posedge clk2) sig1 ##1 sig2;
endproperty
```

```
property p4;
@(posedge clk2) s2; //illegal
endproperty
```

The last property is invalid because it instantiates a sequence with a different clock tick than the clock tick for the property. In the valid properties you see the following:

- How to specify the clock tick in the property for the sequence it instantiates.
- How to specify a clock tick both in the property and in the sequence instantiated in the property, if they specify the same clock tick.
- That instead of instantiating a sequence you can include a sequential expression like `sig1 ##1 sig2`.

You can declare a property in the following places in your code:

- In a module definition
- In an interface definition
- At `$root` (in SystemVerilog `$root` means outside of any other definition, a property defined in `$root` is globally accessible).

Note:

The SystemVerilog LRM says that you can declare a property in a module definition but never in an `always` or `initial` block.

Using Formal Arguments in a Property

Like sequences, you can include formal arguments in properties. Unlike sequences you cannot use or instantiate a property in another property. You use or instantiate a property in a concurrent assertion. For example:

```
property p3 (sig1,sig2);
@(posedge clk2) sig1 ##1 sig2;
endproperty
```

```
a1: assert property (p3(sig3,sig4));
```

Here the property uses signals sig1 and sig2 in its sequential expression, but signals sig3 and sig4 replace them in the assertion declaration.

Implications

Property implications, contain a boolean or sequential expression, called an antecedent, which must be true or match before VCS starts to monitor the events in another sequence or sequential expression, called the consequent, to see if that sequence matches.

There are two types of implications:

- Overlapping implications where the antecedent and the first event in the declared sequence or sequential expression happen during the same clock tick. For overlapping implications you enter the `| ->` operator between the antecedent and the consequent.
- Non-overlapping implications where there is a clock tick delay between the antecedent and the consequent. For non-overlapping implications you enter the `| =>` operator between the antecedent and the consequent.

The following are examples of the VCS implemented implication constructs:

```
sequence s1;  
sig1 ##1 sig2;  
endsequence
```

```
property p1;  
@(posedge clk) (sig3 || sig4) |-> s1;  
endproperty
```

Property p1 contains an overlapping implication. It specifies checking that `(sig3 && sig4)` is true and if so, during the same clock tick, checking to see if `sig1` is true, and then, a clock tick later, checking to see if `sig2` is true.

```
property p2;  
@(posedge clk) (sig1 ##1 sig2) |-> (sig3 ##1 sig4);  
endproperty
```

Property p2 also contains an overlapping implication. In p2 the antecedent is a sequential expression.

```
property p3;  
@(posedge clk) (sig3 ##1 sig4) |=> ##1 (sig1 ##1 sig2);  
endproperty
```

Property p3 contains a non-overlapping implication. The first event is the sequential expression, `sig1` being true, must happen one clock tick after the antecedent expression is true.

Remember that a property is either true or false, so for a property to be true, by default the antecedent must be true and the consequent must succeed. If you include the keyword `not` between the implication operator and the consequent, the property is true if the consequent does not succeed, for example:

```
property p4;  
@(posedge clk) (sig3 && sig4) |-> not (sig1 ##1 sig2);  
endproperty
```

Property p4 is true if, when `(sig3 && sig4)` is true, `sig1` is not true, or if it is, one clock tick later, `sig2` is not true.

Inverting a Property

The keyword `not` can also be used before the declared sequence or sequential expression, or if it contains an implication, before the antecedent, to make the property true, if it otherwise would be false, or make the property false if it otherwise would be true. For example:

```
sequence s1;
sig1 ##1 sig2;
endsequence

property p1;
@(posedge clk) not s1;
endproperty
```

Property p1 is true if sig1 is never true, or if it is, one clock tick later sig2 is never true.

```
sequence s2;
@(posedge clk2) sig4 ##1 sig5;
endsequence

property p2;
not s2;
endproperty
```

Property p2 is true if sig4 is never true, or if it is, one clock tick later sig5 is never true.

```
property p3;
@(posedge clk) not (sig3 && sig4) |-> not sig1 ##1 sig2;
endproperty
```

Property p3 is true if the implication antecedent is never true, or if it is, the consequent sequential expression succeeds. Notice here that the keyword `not` occurs twice in the property.

Past Value Function

SystemVerilog has a `$past` system function that returns the value of a signal from a previous clock tick. The following is an example of its use:

```
property p1;
@(posedge clk) (cnt == 0) ##3 ($past(cnt,3)==0);
endproperty
```

This rather elementary use of the `$past` system function returns the value of signal `cnt` from three clock ticks ago. The first argument, an expression, is required. The second argument, a number of clock ticks previous to the current clock tick, is optional and defaults to 1.

The disable iff Construct

The `disable iff` construct enables the use of asynchronous resets. It specifies a reset condition in which all attempts that have started for properties immediately succeed, and all subsequent attempts succeed as soon as they start.

The following shows a use of the `disable iff` construct:

```
initial
begin
  clk=0;
  rst=0;
  sig1=1;
  #7 sig2=1;
  :
end
always
#5 clk=~clk;

sequence s1;
```



```

sig1 ##1 sig2;
endsequence

property p1;
@(posedge clk) disable iff (rst) s1;
endproperty

a1: assert property (p1);

```

If during simulation sig2 turns false, the property no longer succeeds. If, some time later, rst turns true, the property starts to succeed again. If rst turns false again, the property once again no longer succeeds.

assert Statements

VCS never checks a property or a sequence unless it is instantiated in a concurrent assertion. The concurrent assertion enforces the property or sequence as a checker of that property or sequence.

A concurrent assertion takes the form of an `assert` statement. The following is an example of an `assert` statement:

```
a1: assert property (p1);
```

In this `assert` statement:

`a1:`

The instance name of the concurrent assertion. Concurrent assertions have hierarchical name beginning with the hierarchical name of the module instance in which they are declared, and ending with this instance name. Instance names are optional.

`assert`

Keyword for declaring a concurrent assertion.

`property`

Keyword for instantiating both a property or a sequence.

`p1`

Property instantiated in the concurrent assertion. You could also have specified a sequence instead of a property.

You can declare a concurrent assertion, and enter an `assert` statement, in the following places in your code:

- In a module definition
- In an Interface definition
- In `$root`

Note:

- In the VCS implementation, you can declare a concurrent assertion in a module definition including inside an `always` block but not in an `initial` block. The Accellera SystemVerilog LRM allows concurrent assertions in `initial` blocks.

If a property has formal arguments you can replace them with other signals as shown in ["Using Formal Arguments in a Property" on page 23-18](#).

assume Statements

The `assume` statement specifies a property that VCS can assume about the simulation environment. As specified in the Accellera language reference manual, an `assume` statement is a hypothesis for proving asserted properties in `assert` statements.

Like an asserted property, VCS checks an assumed property and reports if the assumed property fails to hold.

`assume` statements are syntactically similar to `assert` statements, as stated in the Accellera document. The biasing feature is only useful when properties are considered as assumptions to drive random simulation. When a property with biasing is used in an assertion or coverage, the list operator is equivalent to inside operator, and the weight specification is ignored. Therefore the following `assume` statement is functionally equivalent to the following `assert` statement:

```
a1:assume property @(posedge clk) req dist {0:=40, 1:=60} ;  
a1_assertion:assert property req inside {0, 1} ;
```

cover Statements

The `cover` statement calls for the monitoring of a property or a sequence. VCS looks for matches, how often the property was true or how often the sequence occurred. When simulation is over, VCS displays the results of this monitoring.

A `cover` statement is syntactically similar to an `assert` statement. The following is an example of a `cover` statement:

```
c1: cover property (p1);
```

In this `cover` statement:

`c1`:

Instance name of the `cover` statement. `cover` statements have hierarchical name beginning with the hierarchical name of the module instance in which they are declared, and ending with this instance name. Instance names are optional.

`cover`

Keyword for declaring a `cover` statement.

`property`

Keyword for instantiating both a property or a sequence.

`p1`

Property instantiated in the cover statement. You could have specified a sequence instead of a property.

The following SVA code contains two `cover` statements:

```
sequence s1;
@(posedge clk) sig1 ##[1:3] sig2;
endsequence

sequence s2;
sig3 ##[1:3] sig4;
endsequence

property p1;
@(posedge clk) sig1 && sig2 |=> s2;
endproperty

a1: assert property (p1);
a2: assert property (@(posedge clk)s1);
c1: cover property (p1);
c2: cover property (@(posedge clk)s1);
endmodule
```

VCS, for example, displays the following after simulation as a result of these `cover` statements:

```
"exp3.v", 31: test.c1, 9 attempts, 16 total match, 7 first match, 1 vacuous match  
"exp3.v", 32: test.c2, 9 attempts, 21 total match, 8 first match, 0 vacuous match
```

This display is explained as follows:

- In the first line:
 - The `cover` statement is in source file `exp3.v`.
 - The instance of the `cover` statement is `test.c1`. It is declared in module `test`.
 - There were nine attempts to cover the property `p1`.
 - In those nine attempts there were 16 times that the property was true. There can be more than one match in a attempt. In this case in property `p1`, in sequence `s2`, a match can occur over a range of clock ticks and in this case more than once in the range.
 - There were seven first matches. The property was true seven times at the start of the range.
 - There was a vacuous match. Property `p1` contains an implication. The antecedent `sig1 && sig2` was false making the implication vacuously true because it doesn't mean that the consequent sequence `s2` occurred.
- In the second line:
 - The `cover` statement is in source file `exp3.v`.
 - The instance of the `cover` statement is `test.c2`. It is declared in module `test`.
 - There were nine attempts to cover the sequence `s1`.

- In those nine attempts there were 21 times that the sequence occurred.
- There were no vacuous matches because the `cover` statement does not instantiate a property with an implication.

You can declare a `cover` statement, in the following places in your code:

- In a module definition
- In an Interface definition
- In `$root`

Note:

In the VCS implementation, you can declare a `cover` statement in a module definition including inside an `always` block but not in an `initial` block. The Accellera SystemVerilog LRM allows `cover` statements in `initial` blocks.

`cover` statements, unlike `assert` statements, only have a pass action block, not a fail action block, in the VCS implementation.

Action Blocks

`assert` statements can have a pass and a fail action block, and `cover` statements can have a pass action block. The pass block executes when the `assert` or `cover` statement succeeds. The fail block, that follows the keyword `else`, executes when the `assert` statement fails. The following are examples of these blocks:

```
a1: assert property (p1)
begin
    $display("p1 succeeds");
    passCount ++;
```

```

end
else
begin
    $display("p1 does not succeed");
    failCount ++;
end

c1: cover property (p1)
begin
    $display("p1 covered");
    coverCount ++;
end

```

Binding An SVA Module To A Design Module

You can define a module that contains just SVA `sequence` and `property` declarations, and `assert` and `cover` statements. The module ports are signals in these declarations and statements. These ports are also signals in a design module (a module that contains behavioral or RTL code or other types of design constructs).

You can then bind the SVA module to the design module and it is the same as instantiating the SVA module in the design module. The following is an example of a design module, and SVA module and a `bind` directive that binds the SVA module to the design module:

```

module dev;
    logic clk,a,b;
    :
endmodule

module dev_checker (input logic CLK, input logic A,
    input logic B);
    property p1;
        @(posedge CLK) A ##1 B;
    endproperty

```

```
a1: assert property(p1) else $display("p1 Failed");
endmodule
```

```
bind dev dev_checker dc1 (clk,a,b);
```

In this `bind` directive:

```
bind
```

Keyword that starts the bind directive

```
dev
```

Module identifier (name) of the module to which you want to bind the SVA module

```
dev_checker
```

Module identifier of the SVA module

```
dc1
```

Instance name of the SVA module.

```
(clk,a,b)
```

Port connection list to the SVA module

You can also bind an SVA module to a design module instances. For example:

```
module top;
  logic clk,a,b;
```

```
  ⋮
```

```
  dev d1 (clk,a,b);
```

```
endmodule
```

```
module dev (input logic clk, input logic a, input logic b);
```

```
  ⋮
```

```
endmodule
```

```
module dev_checker (input logic clk, input logic a, input
  logic b);
```



```

property p1;
  @(posedge clk) a ##1 b;
endproperty

a1: assert property(p1) else $display("p1 Failed");
endmodule

bind top.d1 dev_checker dc1 (clk,a,b);

```

In this `bind` directive `top.d1` is an instance of module `dev`.

IMPORTANT:

Binding to an instance that is generated using a `generate` statement is not supported.

Parameter Passing In A bind Directive

The module containing the SVA code that is bound to the design module is instantiated in the `bind` directive and as such you can use parameter passing in the instantiation. The following is an example:

```

`timescale 1ns/1ns
module dev;
  logic clk,sig1,sig2;
  :
endmodule

module dev_check(input logic CLOCK,input logic SIG1, input
  logic SIG2);
  parameter ticks =5;

  property p1;
    @(posedge CLOCK) SIG1 ##ticks SIG2;
  endproperty

  a1: assert property(p1) else $display("\n\np1 failed\n\n");
endmodule

```

```
bind dev dev_check #(10) dc1 (clk,sig1,sig2);
```

Notice that module `dev_check`, that contains the SVA code, also has a parameter for specifying the number of clock ticks in the property. In the parameter declaration it has a value of 5, but its value is changed to 10 in the `bind` directive, like in any other module instantiation.

The VPI For SVA

This VPI is to enable you to write applications that react to SVA events and to enable you to write SVA waveform, coverage, and debugging tools.

Note:

To use this API you need to include the `sv_vpi_user.h` file along with the `vpi_user.h` file in the `$VCS_HOME/include` directory. See section 28 of the *SystemVerilog 3.1a LRM*.

This section describes the differences between the VCS implementation and section 28 of *SystemVerilog 3.1a LRM*.

- In subsection 28.3.2.2 “Extending `vpi_get()` and `vpi_get_str`,” use `vpiDefFileName` instead of `vpiFileName`.
- In subsection 28.4.1 “Placing assertion system callbacks,” `cbAssertionSysStop` is not supported.
- In subsection 28.4.2 “Placing assertion callbacks,” the `failEpr` step information is not supported.

- In subsection 28.5.1 “Assertion system control,” `vpiAssertionSysStart` and `vpiAssertionSysStop` are not supported.
- In subsection 28.5.2 “Assertion control,” `vpiAssertionDisableStep` and `vpiAssertionEnableStep` are not supported.

Also the following, in the static part, are not supported:

- Assignments inside sequence expressions
- Bit-selects or part-selects of formal arguments inside a sequence expression

SystemVerilog Assertion Local Variable Debugging

VCS includes four callback types that you can use to debug SVA local variables. You register these callback types on a VPI assertion handle using the `vpi_register_assertion_cb` method.

These callback types are as follows:

`CreationcbAssertionLocalVarCreated`

VCS calls this callback type when VCS creates the SVA local variable. This happens when a new SVA attempt starts.

`cbAssertionLocalVarUpdated`

VCS calls this callback type when it updates an SVA local variable. The new value might be the same as the old value.

`cbAssertionLocalVarDuplicated`

VCS calls this callback type when it duplicates an SVA local variable. This happens when an attempt forks off into multiple paths.

cbAssertionLocalVarDestroyed

VCS calls this callback type when it destroys an SVA local variable. This happens when an assertion succeeds or fails.

All these callback types return a handle to the local variable that caused the event. Use the handle provided in the callback to get the name and value of the local variable. Your application is responsible for keeping track of the destroyed local variable handles. Using a destroyed variable handle results in unpredictable behavior.

You use the `vpi_register_assertion_cb` method to register a callback on an SVA. Whenever a local variable event happens on that SVA, VCS invokes the corresponding callback with that local variable handle embedded in the attempt information.

Note:

You do not have the flexibility to register a callback on a single local variable or part (bit or part select) of the variable. For embedding the local variable handle the `attempt_info` structure is extended as follows.

```
typedef struct t_vpi_attempt_info {
    union {
        vpiHandle failExpr;
        p_vpi_assertion_step_info step;
        p_vpi_attempt_local_var_info local_var_info;
    } detail;
    s_vpi_time attemptStartTime; /* Time attempt triggered */
} s_vpi_attempt_info, *p_vpi_attempt_info;

typedef struct t_vpi_attempt_local_var_info {
    vpiHandle localVar;
} s_vpi_attempt_local_var_info,
*p_vpi_attempt_local_var_info;
```

Note that if VCS duplicates the SVA local variable, the returned `vpi_attempt_info` structure contains the handle to the new local variable. Your application needs to keep track of all copies of local variable for a particular attempt.

These callback types have the following limitations:

- The name or fullname of the SVA local variable does not contain the name of sequence or property it is declared in.
- The local variable handle supports only `vpiType`, `vpiName`, `vpiFullName` and `getValue`. There is no support for other properties defined on normal VPI variables.
- VCS treats XMR (cross module reference) SVA local variables as normal SVA local variables, so you cannot get the XMR part in the local variable name. `vpiFullName` considers the sequence or property instantiated as if it is declared in the scope containing the assertion.
- Change in part of a variable, for example a one bit change in a vector results in a callback of the full variable. Your application is responsible for identifying the changed part.
- No debug support for structure/union/classes in the initial implementation. Your application can get callbacks on other local variables in the assertion. VCS ignores only unsupported callback types.

Controlling How VCS Uses SystemVerilog Assertions

You use compile-time and runtime options to control SystemVerilog in VCS. Together with system tasks, these options allow you to use all the features described in the following sections.

Compile-Time And Runtime Options

VCS has the following compile-time option for controlling SystemVerilog assertions:

```
-assert keyword_argument
```

The keyword arguments are as follows:

```
enable_diag
```

Enables further control of results reporting with runtime options.

```
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. So the first sampling point and subsequent sampling points should be ignored until the past buffer has been filled with respect to the sampling point. Using this keyword filters out vacuous successes too.

```
disable
```

Disables all SystemVerilog assertions in the design.

```
disable_cover
```

When you include the `-cm assert` compile-time and runtime option, VCS include 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. See ["Disabling SystemVerilog Assertions at Compile-Time" on page 23-42](#).

`dumpoff`

Disables the dumping of SVA information in the VPD file during simulation.

VCS has the following runtime option for controlling SystemVerilog assertions:

`-assert keyword_argument`

The keyword arguments are as follows:

`dumpoff`

Disables the dumping of SVA information in the VPD file during simulation.

`dve`

Tells VCS to record SystemVerilog assertion information in the VPD file.

`filter`

Blocks reporting of trivial implication successes. These happen when an implication construct registers a success only because the precondition (antecedent) portion is false (and so the consequent portion is not checked). With this option, reporting only shows successes in which the whole expression matched.

`finish_maxfail=N`

Used for simulation control, terminates the simulation if the number of failures for any assertion reaches *N*. *N* must be supplied, otherwise no limit is set.

`global_finish_maxfail=N`

Used for simulation control, stops the simulation when the total number of failures, from all SystemVerilog assertions, reaches *N*.

`maxcover=N`

When you include the `-cm assert` compile-time and runtime option, VCS include information about cover statements in the assertion coverage reports. This argument 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 can't be 0.

`maxfail=N`

Limits the number of failures for each assertion to N . When the limit is reached, the assertion is disabled. N must be supplied, otherwise no limit is set.

`maxsuccess=N`

Limits the total number of reported successes to N . N must be supplied, otherwise no limit is set. The monitoring of assertions continues, even after the limit is reached.

`nocovdb`

When you include the `-cm assert` compile-time and runtime option, VCS records assertion coverage information in the `./simv.vdb/fcov/results.db` database file. This argument tells VCS not to write the this file.

`nopostproc`

Whether or not you include the `-cm assert` compile-time and runtime option, after simulation VCS displays the SystemVerilog assertion coverage summary. This argument disables the display of this summary. This summary looks like this 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 this file's name and location is `./assert.report`, but you can change it to where you want by entering the filename path name argument.

The filename can start with a number or letter. The following special characters are acceptable in the filename: `%`, `^`, and `@`. Using the following unacceptable special characters: `#`, `&`, `*`, `[]`, `$`, `()`, or `!` has the following consequences:

- A filename containing `#` or `&` results in a filename truncation to the character before the `#` or `&`.
- A filename containing `*` or `[]` results in a no match message.
- A filename containing `$` results in an undefined variable message.
- A filename containing `()` results in a badly placed `()`'s message.
- A filename containing `!` results in an event not found message.

`success`

Enables reporting of successful matches, and successes on `cover` statements, in addition to failures. The default is to report 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.

You can enter more than one keyword, using the plus `+` separator. For example:

```
-assert maxfail=10+maxsuccess=20+success+filter
```

By default VCS collects coverage information on `cover` statements, you can limit or disable collecting this information with the `maxcover` and `nocovdb` arguments.

VCS collects coverage information on `assert` statements when you enter the `-cm assert` compile-time option and keyword argument.

Most of the compile-time and runtime options that were implemented for OpenVera assertions also work on SystemVerilog assertions. These compile-time options are as follows:

```
-ova_cov           -ova_cov_events   -ova_cov_hier
-ova_debug         -ova_dir          -ova_file
-ova_filter_past   -ova_enable_diag
```

These runtime options are as follows:

```
-ova_quiet         -ova_report       -ova_verbose
-ova_filter        -ova_max_fail     -ova_max_success
```

```
-ova_simend_max_fail      -ova_success      -ova_cov
-ova_cov_name             -ova_cov_db
```

See ["Compiling Temporal Assertions Files"](#) on page 20-19, ["OVA Runtime Options"](#) on page 20-21, and ["Functional Code Coverage Options"](#) on page 20-24.

Ending Simulation at a Number of Assertion Failures

There are two ways to end simulation when the total number of failures from all assertions reaches a specified number:

- Using the `-assert global_finish_maxfail=N` runtime option and argument, see ["Compile-Time And Runtime Options"](#) on page 23-36.
- Using the `$ova_set_global_finish_maxfail` system task.

The `$ova_set_global_finish_maxfail` takes an argument which is an expression. For example:

```
$ova_set_global_finish_maxfail(100);
```

This expression does not need to be a constant expression. For example:

```
$ova_set_global_finish_maxfail(reg1 + reg2);
```

When VCS executes this system task, the current value of the expression argument determines the total number of assertion failures that ends simulation.

Disabling SystemVerilog Assertions at Compile-Time

You can specify a list of SystemVerilog assertions in your code that you want to disable at compile-time. You do so with the `-assert` compile-time option and `disable_file=filename` argument, for example:

```
vcs -sverilog -assert disable_file=disable_assertions.txt
```

Enter one absolute hierarchical name of a SystemVerilog assertion on each line, for example:

```
test.dev1.a1
```

Only one hierarchical assertion name to a line.

Entering SystemVerilog Assertions as Pragmas

If your code has to be read by a tool that has not implemented SystemVerilog Assertions, you can enter your SVA code as pragmas (or metacomments) so that the other tool ignores the SVA code. You can tell VCS to compile the SVA code by including the `-sv_pragma` compile-time option. The following is an example of SVA code as pragmas:

```
// sv_pragma sequence s1;
// sv_pragma @(posedge clk) sig1 ##[1:3] sig2;
// sv_pragma endsequence

/* sv_pragma
sequence s2;
sig3 ##[1:3] sig4;
endsequence

property p1;
```

```

@(posedge clk) sig1 && sig2 => s2;
endproperty

a1: assert property (p1);
a2: assert property (@(posedge clk)s1);
c1: cover property (p1);
c2: cover property (@(posedge clk)s1);
*/

```

The `sv_pragma` keyword must immediately follow the characters that begin the comment: `//` for single line comments and `/*` for multi-line comments.

Note:

This feature is intended allow SVA code as pragmas. When you include the `-sv_pragma` compile-time option, VCS compiles all the contents in the comment, not just the SVA code in the comment. If the multi-line comment is the following:

```

/* sv_pragma
a1: assert property (p1);
a2: assert property (@(posedge clk)s1);
c1: cover property (p1);
c2: cover property (@(posedge clk)s1);

initial
$display("$display with SVAs");
*/

```

VCS displays the `$display with SVAs` character string at runtime.

Options for SystemVerilog Assertion Coverage

SystemVerilog assertion coverage monitors the design for when assertions are met and not met. Coverage results are on assertions, not the properties or sequences that might be instantiated in these assertions. See ["Reporting On Assertions Coverage" on page 23-45](#)

To enable and control assertion coverage, VCS has the following compile-time options:

`-cm assert`

Compiles for SystemVerilog assertions coverage. `-cm` is not a new compile-time option but the `assert` argument is new. This option and argument must also be entered at runtime.

`-cm_assert_hier filename`

Limits assertion coverage to the module instances specified in *filename*. Specify the instances using the same format as VCS coverage metrics. If this option is not used, coverage is implemented on the whole design.

There are also the following runtime options for assertion coverage:

`-cm assert`

Specifies monitoring for SystemVerilog assertions coverage. Like at compile-time, `-cm` is not a new runtime option but the `assert` argument is new.

`-cm_assert_name path/filename`

Specifies the path and filename 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`.

`-cm_assert_report path/filename`

Specifies the file name or the full path name of the assertion coverage report file. This option overrides the default report name and location, which is `./simv.vdb/fcov/results.db`. If only a file name is given, the default location is used resulting in: `./simv.vdb/fcov/filename.db`.

VCS also has the following runtime option and keyword options for assertion coverage:

`-assert nocovdb`

Tells VCS not to write the `results.db` database file for assertion coverage. Without this file there is no coverage data.

Reporting On Assertions Coverage

After running a series of simulations, you can generate a report summarizing the coverage of both kinds of assertions. With this report, you can quickly see if all assertions were attempted, how often they were successful, and how often they failed. Potential problem areas can be easily identified. The report can cover one test or merge the results of a test suite. The report is written in HTML and you can customize it with a Tcl script.

The default report shows the number of assertions that:

- Were attempted
- Had successes
- Had failures

Coverage is broken down by module and instance, showing for each assertion and expression, the number of attempts, failures, and successes.

Assertion coverage can also grade the effectiveness of tests, producing a list of the minimum set of tests that meet the coverage target. Tests can be graded on any of these metrics:

- Number of successful assertion attempts versus number of assertions (*metric = SN*)
- Number of failed assertion attempts versus number of assertions (*metric = FN*)
- Number of assertion attempts versus number of assertions (*metric = AN*)
- Number of successful assertion attempts versus number of assertion attempts (*metric = SA*)
- Number of failed assertion attempts versus number of assertion attempts (*metric = FA*)

To generate a report, run the following command:

```
assertCovReport [options]
```

The command line options are as follows:

```
-e TCL_script | -
```

Use this option to produce a custom report using Tcl scripts or entering Tcl commands at standard input (keyboard). Most of the other `assertCovReport` options are processed before the Tcl scripts or keyboard entries.

```
-e TCL_script
```

Specifies the path name of a Tcl script to execute. To use multiple scripts, repeat this option with each script's path name. They are processed in the order listed.

```
-e -
```

Specifies your intent to enter Tcl commands at the keyboard.

The Tcl commands provided by VCS, that you can input to `fcovReport` for OVA coverage reports (see ["Tcl Commands For SVA And OVA Functional Coverage Reports" on page 23-49](#)), you can also input to `assertCovReport` for SystemVerilog assertion (SVA) coverage.

- `-cm_assert_cov_cover`
Specifies reporting only about `cover` statements.
- `-cm_assert_cov`
Specifies reporting only about `cover` and `assert` statements (no OVA coverage).
- `-cm_assert_category category_val`
`[, category_val...]`
Reports only on assertions specified by category value. You can specify any number of category values, separating them with commas.
- `-cm_assert_cov_events`
Specifies only reporting on OpenVera assertion events.
- `-cm_assert_name path`
Specifies the path of the template database. If this option is not included, `assertCovReport` uses `simv.vdb`.
- `-cm_assert_grade_instances target, metric`
`[, time_limit]`
Generates an additional report, `grade.html`, that lists the minimum set of tests that add up to the target value for the metric (see previous page for *metric* codes). The grading is by instance.
- `-cm_assert_grade_module target, metric`
`[, time_limit]`
Generates an additional report, `grade.html`, that lists the minimum set of tests that add up to the target value for the metric (see previous page for *metric* codes). The grading is by module.

`-cm_assert_map filename`

Maps the module instances of one design onto another while merging the results. For example, use this to merge the assertion coverage results of unit tests with the results of system tests. Give the path name of a file that lists the hierarchical names of from/to pairs of instances with one pair per line:

```
from_name to_name
```

The results from the first instance are merged with the results of the second instance in the report.

`-cm_assert_merge filename`

Specifies the path name of an assertion coverage result file or directory to be included in the report. If *filename* is a directory, all coverage result files under that directory are merged. Repeat this option for any result file or directory to be merged into this report. If this option is not used, `assertCovReport` merges all the result files in the directory of the template database (specified with `-cm_assert_dir` or `simv.vdb/snps/fcov` by default).

`-cm_assert_report name | path/name`

Specifies the base name for the report. The `assertCovReport` command creates an HTML index file at `simv.vdb/reports/name.fcov-index.html` and stores the other report files under `simv.vdb/reports/name.fcov`.

If you give a path name, the last component of the path is used as the base name. So the report files are stored under `path/name` and the index file is at `path/name.fcov-index.html`.

If this option is not included, the report files are stored under `simv.vdb/reports/report.fcov` and the index file is named `report.fcov-index.html`.

```
-cm_assert_severity int_val [,int_val...]
```

Reports only on OpenVera assertions specified by severity value. You can specify any number of severity values, separating them with commas. Only OpenVera assertions can have a severity, SystemVerilog assertions cannot.

Tcl Commands For SVA And OVA Functional Coverage Reports

You can produce a custom report with Tcl commands that you enter in assertCovReport. These Tcl commands also work in fcovReport when you want a custom report about OVA coverage. These commands are in Table 23-1.

The Tcl command descriptions frequently refer to a bin. A bin is a coverage value container in the coverage database. There are two kinds of bins: boolean (which hold 0 or 1 values) and count. Count bins can hold any unsigned value that can be stored in 32 bits.

Table 23-1 Tcl Commands for SVA and OVA Functional Coverage Reports

Command	Return Value	Description
<code>fcov_get_assertions -instance <i>handle</i></code>	array of handles	Returns an array of handles to the assertions of the instance with the specified handle.
<code>fcov_get_assertions -module <i>handle</i></code>	array of handles	Returns an array of handles to the assertions of the module with the specified handle.
<code>fcov_get_bins -assertion <i>handle</i></code>	array of handles	Returns an array of handles to the bins of the assertion with the specified handle.
<code>fcov_get_category -assertion <i>handle</i></code>	int	Returns the category of the assertion with the specified handle expressed as an integer.

Command	Return Value	Description
<code>fcov_get_children -instance <i>handle</i></code>	array of handles	Returns an array of handles for the child instances that contain at least one assertion under their hierarchy. The parent instance is with the specified handle.
<code>fcov_get_coverage -bin <i>handle</i></code>	int	Returns the coverage count associated with the bin with the specified handle. It can be 0 or the count of the times the bin was covered.
<code>fcov_get_flag -bin <i>handle</i></code>	string	Returns the flag associated with the bin with the specified handle. It can be: "illegal", "ignored", or "impossible".
<code>fcov_get_handle -instance <i>name</i></code>	handle	Returns a handle for the instance specified with its full hierarchical name.
<code>fcov_get_handle -module <i>name</i></code>	handle	Returns a handle for the specified module.
<code>fcov_get_handle -instance -module <i>name1</i> -assertion <i>name2</i></code>	handle	Returns the handle to the specified assertion in the specified instance or module. The assertion name must follow the convention for the full name of an assertion.
<code>fcov_get_handle -instance -module <i>name1</i> -assertion <i>name2</i> -bin <i>name3</i></code>	handle	Returns the handle to the specified bin for the specified assertion in the specified instance or module. Current names are: " 1attempts", " 2failures", " 3allsuccesses", " 4realsuccesses", and " 5events" (note that all bin names start with a space).
<code>fcov_get_instances</code>	array of handles	Returns an array of handles for the instances that contain at least one assertion under their hierarchy.
<code>fcov_get_instances -module <i>handle</i></code>	array of handles	Returns an array of handles for the instances that contain at least one assertion under their hierarchy for the module with the specified handle.

Command	Return Value	Description
<code>fcov_get_modules</code>	array of handles	Returns an array of handles for the modules that contain at least one assertion.
<code>fcov_get_name -object <i>handle</i></code>	string	Returns the name of the object the specified handle.
<code>fcov_get_no_bins -assertion <i>handle</i></code>	int	Returns total bins for the assertion with the specified handle.
<code>fcov_get_no_of_assertions</code>	int	Returns total number of assertions in the design..
<code>fcov_get_no_of_assertions -instance <i>handle</i></code>	int	Returns total number of assertions for the instance with the specified handle.
<code>fcov_get_no_of_assertions -module <i>handle</i></code>	int	Returns total number of assertions for the module with the specified handle.
<code>fcov_get_no_of_assertions_attempted</code>	int	Returns total number of assertions attempted in the design.
<code>fcov_get_no_of_assertions_attempted -instance <i>handle</i></code>	int	Returns total number of assertions attempted for the instance with the specified handle.
<code>fcov_get_no_of_assertions_attempted -module <i>handle</i></code>	int	Returns total number of assertions attempted for the module with the specified handle.
<code>fcov_get_no_of_assertions_failed</code>	int	Returns total number of assertions that failed in the design.
<code>fcov_get_no_of_assertions_failed -instance <i>handle</i></code>	int	Returns total number of assertions that failed for the instance with the specified handle.
<code>fcov_get_no_of_assertions_failed -module <i>handle</i></code>	int	Returns total number of assertions that failed for the module with the specified handle.
<code>fcov_get_no_of_assertions_succeeded</code>	int	Returns total number of assertions that succeeded/matched in the design.
<code>fcov_get_no_of_assertions_succeeded -instance <i>handle</i></code>	int	Returns total number of assertions that succeeded/matched for the instance with the specified handle.
<code>fcov_get_no_of_assertions_succeeded -module <i>handle</i></code>	int	Returns total number of assertions that succeeded/matched for the module with the specified handle.

Command	Return Value	Description
<code>fcov_get_no_of_children -bin <i>handle</i></code>	int	Returns total number of child bins whose parent is the bin handle handle.
<code>fcov_get_no_of_children -instance <i>handle</i></code>	int	Returns total number of child instances containing at least one assertion under their hierarchy. The parent instance is with the specified handle.
<code>fcov_get_no_of_instances</code>	int	Returns total number of instances that contain at least one assertion under their hierarchy.
<code>fcov_get_no_of_instances -module <i>handle</i></code>	int	Returns total number of instances containing at least one assertion under their hierarchy for a module with the specified handle.
<code>fcov_get_no_of_modules</code>	int	Returns total number of modules that contain at least one assertion.
<code>fcov_get_no_of_topmodules</code>	int	Returns total number of top level modules that contain at least one assertion under their hierarchy. (Top level modules are instances.)
<code>fcov_get_severity -assertion <i>handle</i></code>	int	Returns the severity of an assertion with the specified handle expressed as an integer.
<code>fcov_get_topmodules</code>	array of handles	Returns an array of handles for the top level modules that contain at least one assertion under their hierarchy. (Top level modules are instances.)
<code>fcov_get_type -bin <i>handle</i></code>	string	Returns the type of the bin with the specified handle. It can be: "bool" or "count".
<code>fcov_get_type -object <i>handle</i></code>	string	Returns the type of the object with the specified handle. The type can be "module", "instance", "assertion", or "bin".

Command	Return Value	Description
<code>fcov_grade_instances -target <i>value1</i> -metric <i>code</i> [-timeLimit <i>value2</i>]</code>	string	Returns a list of the minimum set of tests that add up to the target value for the metric code. Each test is accompanied by the accumulated coverage value including that test. The grading is by instance.
<code>fcov_grade_modules -target <i>value1</i> -metric <i>code</i> [-timeLimit <i>value2</i>]</code>	string	Returns a list of the minimum set of tests that add up to the target value for the metric code. Each test is accompanied by the accumulated coverage value including that test. The grading is by module.
<code>fcov_is_cover_prop -assertion <i>handle</i></code>	int	Returns 1 if the assertion is the cover directive for the property.
<code>fcov_is_cover_seq -assertion <i>handle</i></code>	int	Returns 1 if the assertion is the cover directive for the sequence.
<code>fcov_load_design -file <i>name</i></code>	empty string ""	Unloads any existing design and data, including all name maps. Then loads a design with the specified path name. (The search rules are described in the table for options.)
<code>fcov_load_test -file <i>name</i></code>	empty string ""	Loads a test with the specified name. The name can be a file name or full path name. (The search rules are described in the table for options.)
<code>fcov_load_test_grade -file <i>name</i></code>	empty string ""	Loads a file with the specified name for coverage grading. The name can be a file name or full path name. (The search rules are described in the table for options.)
<code>fcov_map_hier -from <i>hier_name</i> -to <i>hier_name</i></code>	empty string ""	Maps coverage of instance (and the hierarchy under it) with the specified hierarchical name <i>hier_name</i> to another instance for all subsequent merges.
<code>fcov_set -bin <i>handle</i> -coverage <i>int</i></code>	empty string ""	Sets the coverage count <i>int</i> for the bin with the specified handle. It can be 0 or the count of the times the bin was covered.

Command	Return Value	Description
<code>fcov_set -bin <i>handle</i> -flag <i>name</i></code>	empty string ""	Sets the flag with <i>name</i> for the bin with the specified handle. The flag can be: "illegal", "ignored", or "none".
<code>fcov_write_coverage -file <i>name</i></code>	empty string ""	Writes current coverage to a file with the specified name. The name can be a file name or full path name. (The search rules are described in the table for options.)

The names of bins in the coverage database are as follows:

For SVA Assertions

`1attempts`

Holds the count of the attempts.

`2failures`

Holds the count of failed attempts.

`3allsuccess`

Holds the count of successful attempts.

`4realsuccess`

Holds the count of attempts with nonvacuous success.

`8incompletes`

Holds the count of unterminated attempts.

For SVA Property Coverage

`1attempts`

Holds the count of attempts.

`2failures`

Holds the count of failed attempts.

`3allsuccess`

Holds the count of successful attempts.

6vacuousuccess

Holds the count of vacuous successes.

8incompletes

Holds the count of unterminated attempts

For SVA sequence coverage

1attempts

Holds the count of attempts.

3allsuccess

Holds the count of successful attempts.

7firstmatches

Holds the count of generated .matched events.

8incompletes

Holds the count of unterminated attempts.

For OVA assertions

1attempts

Holds the count of attempts.

2failures

Holds the count of failed attempts.

3allsuccess

Holds the count of successful attempts.

4realsuccess

Holds the count of attempts with nonvacuous success.

8incompletes

Holds the count of unterminated attempts.

For OVA events

1attempts

Holds the count of attempts.

2failures

Holds the count of failed attempts.

3allsuccess

Holds the count of successful attempts.

4realsuccess

Holds the count of attempts with nonvacuous success.

5events

Holds the count of generated event matches.

8incompletes

Holds the count of unterminated attempts.

The assertCovReport Report Files

When you compile for SystemVerilog or OpenVera assertion coverage, by including the `-cm assert` compile-time option and keyword argument, VCS creates the `simv.vdb` directory in the current directory.

When you monitor for SystemVerilog or OpenVera assertion coverage, by including the `-cm assert` runtime option and keyword argument, VCS creates the `fcov` directory in the `simv.vdb` directory. This directory contains binary data about SystemVerilog assertions coverage (and if also included, about OpenVera assertions coverage).

When you run `assertCovReport`, this utility does the following:

1. Creates the `reports` directory in the `simv.vdb` directory and writes the `report.index.html` file in the `reports` directory.

2. Creates the report.fcov directory in the reports directory and writes in the report.fcov directory the following files: category.html, hier.html, and tests.html.

Note:

If you include the `-cm_assert_report name` option on the `assertCovReport` command line, you see the following differences:

- The report.fcov directory is named *name.fcov*
- The report.index.html file is named *name.index.html*

The report.index.html File

This file begins with tables of numerical data about the assertions in your design. This information has a title for a type of information about your assertions with a total number and percentage under it. In most cases this title is blue and is a hypertext link to a list of these assertions further down in this file. These titles and what the values under them mean are as follows:

Total number of Assertions

The total number of SystemVerilog `assert` statements and OpenVera `assert` directives in your design.

Assertions not Covered

The total number and percentage of the SystemVerilog `assert` statements for properties that never matched and the OpenVera `assert` directives for sequential expressions that never occur during simulation.

Assertions with at least 1 Real Success

SystemVerilog and OpenVera assertions can fail at certain times during simulation and succeed at other times. This is the total number and percentage of SystemVerilog and OpenVera assertions that had at least one success during simulation.

Assertions with at least 1 Failure

The total number and percentage of SystemVerilog and OpenVera assertions that had at least one failure during simulation.

Assertions with at least 1 Incomplete

Assertions specify a property about the design that can occur over a span of simulation time. This is the total number and percentage of assertions that VCS began to monitor, but simulation ended before the design matched the behavior specified in the assertion.

Assertions without Attempts

VCS looks to see if the design matches the behavior in the assertion when there is a transition on a clock signal for the property (you can specify the type of transition). If none of these clock signal transitions occur, then there is no attempt at the assertion. This is the total number and percentage of assertions with no attempts.

Total number of Cover Directives for Properties

The argument to a SystemVerilog assertion `cover` statement can be the name of a defined and declared property (between the `property` and `endproperty` keywords) or the argument can be just the building blocks of a property between parentheses (a clock signal and a sequential expression).

This value is the total number of SystemVerilog assertions `cover` statements with the name of a property as their argument.

Cover Directive for Property Not Covered

The total number and percentage of SystemVerilog assertion `cover` statements with the name of a property is their argument, where the design's behavior never matches the property specified in the `cover` statement.

Cover Directive for Property with Matches

The total number and percentage of SystemVerilog assertion `cover` statements with the name of a property as their argument, where the design's behavior, at least some of the simulation time, matches the property specified in the `cover` statement.

Cover Directive for Property with Vacuous Matches

A property for a SystemVerilog `cover` statement automatically matches if there is an implication for the property and the antecedent condition is never true. For example:

```
sequence s5;
sig11 ##1 sig12;
endsequence

property p5;
@(posedge clk1) (sig9 ##1 sig10) |-> s5;
endproperty

c4: cover property (p5);
```

If the antecedent `(sig9 ##1 sig10)` never occurs, property `p5` matches, even if the consequent sequential expression `sig11 ##1 sig12` never occurs. This is why such a match is vacuous or empty.

This information is the total number and percentage of SystemVerilog assertion `cover` statements with the name of a property as their argument, and the total number of OpenVera `cover` directives, where there is a vacuous match.

Total number of Cover Directives for Sequences

A SystemVerilog `cover` statement can have a sequence name for an argument instead of a property. For example:

```
sequence s8;  
@(posedge clk1) sig17 ##[1:5] sig18;  
endsequence
```

```
c7: cover property (s8);
```

or

```
sequence s8;  
sig17 ##[1:5] sig18;  
endsequence
```

```
c7: cover property (@(posedge clk1) s8);
```

This information is the total number of such `cover` statements.

Cover Directive for Sequence not Covered

Total number and percentage of SystemVerilog `cover` statements, where the argument is a sequence that did not occur during simulation.

Cover Directive for Sequence with All Matches

If there is a cycle delay range expression in a sequence. For example:

```
sequence s1;  
@(posedge clk1) sig1 ##[1:5] sig2;  
endsequence
```

After `sig1` is true, there could be a match after one, two, three, four, or five rising edges on `clk1`, if `sig2` is also true. If all these matches happen for all attempts, then this is a sequence of all matches.

This information is the total number and percentage of SystemVerilog `cover` statements with sequences with all matches.

Cover Directive for Sequence with First Matches

Total number and percentage of SystemVerilog `cover` statements where the argument is a sequence, and there was a cycle delay range expression, and the first possible match for the sequence occurred.

Total number of Events

In OpenVera assertions, you can define a sequential expression as an event. This information is the total number of such events.

Events Not Covered

Total number and percentage of OpenVera events that did not occur (not covered).

Events with at least 1 real Match

Total number and percentage of OpenVera events that occurred (were covered).

Events without any match or with only vacuous match

Total number and percentage of OpenVera events for which there was no match or only a vacuous match.

Events without any Attempts

Total number and percentage of OpenVera events for which there were no attempts.

At the bottom of the file are hypertext links to the other files written by `assertCovReport`:

- [Lists of tests merged to generate this report \(tests.html\)](#)
- [Hierarchical coverage report \(hier.html\)](#)
- [Category based coverage report \(category.html\)](#)

These links are followed by general information about generating this report.

The tests.html File

You can use the `-cm_assert_map` and `-cm_assert_merge` command line options to merge the results from one design to another. This file indicates the path names of the `results.db` files from which `assertCovReport` merges the results.

The category.html File

You can use the `$ova_set_category` and `$ova_set_severity` system tasks to set the category and severity of a SystemVerilog assertion and an OpenVera assertion.

You can also use the `(* category=number *)` SystemVerilog attribute to set the category of a SystemVerilog assertion.

This file shows you the assertions in your design according to category and severity.

The report begins with categories you used for the SystemVerilog `assert` statements or OpenVera `assert` directives. It shows you the total number and percentage of each category number (integer). The word `Category` preceding a category number is a hypertext link to a list of `assert` statements or directives with that category number, showing you the hierarchical name, number of attempts, successes, failures, and incompletes.

Next is the same information for the severity numbers you assigned to your SystemVerilog `assert` statements or OpenVera `assert` directives. The word `Severity` is a hypertext link to a similar list of `assert` statements or directives with that severity number.

Next is the same information for the category numbers you used for the SystemVerilog `cover` statements where the argument is a property, and the OpenVera `cover` directives.

Next is the same information for the severity numbers you used for the SystemVerilog `cover` statements where the argument is a property, and the OpenVera `cover` directives.

Next is the category numbers you used for the SystemVerilog `cover` statements where a sequence is the argument.

Next is the severity numbers you used for the SystemVerilog `cover` statements where a sequence is the argument.

The hier.html File

The report begins with a list of the module instances in the design and the number of the following in the instances:

- The number (integer) of SystemVerilog `assert` statements or OpenVera `assert` directives.
- The number (integer) of SystemVerilog `cover` statements where the argument is a sequence.
- The number (integer) of SystemVerilog `cover` statements where the argument is a property, and OpenVera `cover` directives.
- The number (integer) of OpenVera events.

Each number is a hypertext link that takes you to a list of each type of statement, directive, or event. For `assert` statements or directives, the list shows you the number of attempts, successes, failures, and incompletes. For `cover` statements or directives and events, the list shows you the number of attempts, all matches, first matches, and incompletes.

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). Its syntax is as follows:

```
$assert_monitor([0|1,] assertion_identifier...);
```

Here:

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.

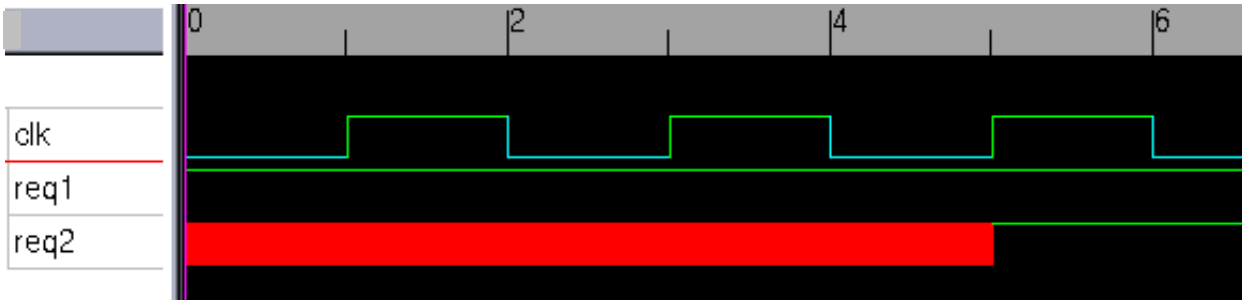
Consider the following assertion:

```
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 startingfound: req1looking for: req2 or
      any
5ns: tracing "test.a1" started at 3ns:
      trace: req1 ##1 anylooking 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 startingfound: req1looking 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 anylooking 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.

Assertion System Functions

The assertion system functions are `$onehot`, `$onehot0`, and `$isunknown`. Their purposes are as follows:

`$onehot`

Returns true if only one bit in the expression is true.

`$onehot0`

Returns true if at the most one bit of the expression is true (also returns true if none of the bits are true).

`$isunknown`

Returns true if one of the bits in the expression has an X value. In the VCS implementation, this function also returns true if one of the bits in the expression has a Z value.

The following is an example of their use:

```
a1: assert property (@ (posedge clk) $onehot({lg1,lg2}));
a2: assert property (@ (posedge clk) $onehot0({lg1,lg3}));
a3: assert property (@ (posedge clk) $isunknown({r1,r2,r3}));
```

Another useful function is `$countones`. This function returns the number of 1s in a bit vector expression.

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]);
```

Here:

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 stop. As seen in ["Stopping And Restarting Assertions By Category" on page 23-71](#), 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]);
```

Here:

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 23-71](#), 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:

```
$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');
```

The categories are masked with the *maskValue* argument and compared with the *categoryValue* argument:

	bits	<i>categoryValue</i>	
category 1	001		
<i>maskValue</i>	1		
result	1	1	match
category 2	010		
<i>maskValue</i>	1		
result	0	1	no match

category 3	011		
<i>maskValue</i>	1		
result	1	1	match
category 4	100		
<i>maskValue</i>	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.

Here is another example. This one 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 `$ova_category_stop` system tasks stop first the odd numbered assertions and then the even numbered ones. The first `$ova_category_stop` system task has a *globalDirective* argument that's 0, the second has a *globalDirective* argument that's 1.
2. The first `$ova_category_start` system task can restart the odd numbered assertions but the second `$ova_category_start` system task can't start the even numbered assertions.

24

SystemVerilog Testbench Constructs

The new version of VCS has implemented some of the SystemVerilog testbench constructs. As testbench constructs they must be in a `program` block (see “Program Blocks” on page 24-15).

Enabling Use of SystemVerilog Testbench Constructs

You enable the use of SystemVerilog testbench constructs with the `-sverilog` compile-time option.

VCS Flow for SVTB

The VCS use model now includes the use model for SystemVerilog NTB. The most important part is analyzing the SystemVerilog source code You can do this as follows:

```
vcs -sverilog -ntb_opts options <SV source code files>
```

As the use of `vcs` indicates, SystemVerilog files are treated like Verilog files in the VCS flow. You can also specify other NTB options:

For example:

```
vcs -sverilog tb.sv
```

or

```
vcs -sverilog -f tb.list
```

Options For Compiling and Simulating SystemVerilog Testbench Constructs

Compile-Time Options

The following compile-time options, used for both Verilog and SystemVerilog code have been tested with SystemVerilog testbench code:

```
-f filename
```

```
+define+macro_name=value
```

```
+incdir+directory_name
```

```
+libext+ext
```

```
-y directory_name
```

```
-timescale=time_unit/time_precision
```


Runtime Options

There are runtime options that were developed for OpenVera testbenches that also work with SystemVerilog testbenches.

```
+ntb_random_seed=integer
```

Sets the seed value used by the top level random number generator at the start of simulation. This option does not work for the Verilog `$random(seed)` system function.

```
+ntb_solver_mode=1|2
```

Specifies the constraint solver mode for the `randomize()` method:

1

The solver spends more pre-processing time in analyzing the constraints, during the first call to `randomize()` on each class. Subsequent calls to `randomize()` on that class are very fast.

2

The solver does minimal pre-processing, and analyzes the constraint in each call to `randomize()`. Default is 2.

The `randomize()` method is described in “Randomize Methods” on page 24-100.

```
+ntb_enable_solver_trace=0|1|2
```

Specifies the debugging mode when VCS executes the `randomize()` method:

0

Disables tracing.

1

Enables tracing. This is the default.

2

Enables tracing with more verbose messages.

The `randomize()` method is described in “Randomize Methods” on page 24-100.

`+ntb_enable_solver_trace_on_failure[=0|1|2]`

Enables a mode that displays trace information only when the constraint solver fails to compute a solution, usually due to inconsistent constraints.

0

Disables tracing.

1

Enables tracing. This is the default. This argument is the default argument when you enter this option without an argument.

2

Enables tracing with more verbose messages and the analysis narrows down to the smallest set of inconsistent constraints, thus aiding the debugging process. This option with the 2 argument is the default condition when you don't enter this option.

Compile Time or Runtime Options

`-cm_dir directory_path_name`

As a compile-time or runtime option, specifies an alternative name and location for the default `simv.vdb` directory, VCS automatically adds the extension `.vdb` to the directory name if not specified.

`-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".

The string Data Type

The string data type is an LCA feature.

VCS has implemented the `string` SystemVerilog data type. The following is the syntax for declaring this data type:

```
string variable_name [=initial_value];
```

String Manipulation Methods

SystemVerilog has the following methods for manipulating strings:

len()

Returns the number of characters in a string.

```
string string_name = "xyz";  
int int1 = string_name.len;
```

getc()

Returns the numerically specified character in the string.

```
bit1=string_name.getc(0);
```

If variable `string_name` has a value of “xyz”, then this method returns the ASCII code for the number 0 character to `bit1`, the x character.

putc()

Replaces a specified character with another value or character. This method takes two arguments, the first is the number of the characters in the string, the second is a value or another string variable. If the second argument is a string value, the specified character is replaced with the first character of the string argument.

```

string string1 = "abc";
string string2 = "xyz";

initial
begin
$display ("string1 is \"%s\"",string1);
string1.putc(0,42);
$display ("string1 is \"%s\"",string1);
string1.putc(1,string2);
$display ("string1 is \"%s\"",string1);
end
endmodule

```

The \$display system tasks display the following:

```

string1 is "abc"
string1 is "*bc"
string1 is "*xc"

```

toupper()

Returns a string with the lower case characters converted to upper case.

```

string string1 = "abc";
string string2 = string1.toupper;
initial
begin
$display("string1 is \"%s\"",string1);
$display("string2 is \"%s\"",string2);
end

```

The \$display system tasks display the following:

```

string1 is "abc"
string2 is "ABC"

```

tolower()

Similar to the toupper method, this method returns a string with the upper case characters converted to lower case.

compare() and icode()

Compares strings and returns 0 if they match, and a value less than 0 or more than zero, depending on the order of the strings, if they don't match. The icode method doesn't see a difference between upper case and lower case.

```
string string1 = "abc";
string string2 = "abc";
string string3 = "xyz";
string string4 = "ABC";

initial
begin
if (string1.compare(string2) == 0)
    $display("string1 matches string2");
if (string1.compare(string3) != 0)
    $display("string1 does not match string3");
if (string1.compare(string4) != 0)
    if (string1.icode(string4) == 0)
        $display("string1 matches string4 except for case");
    else
        $display("string1 does not match string4");
end
```

The \$display system tasks display the following:

```
string1 matches string2
string1 does not match string3
string1 matches string4 except for case
```

substr()

Returns a substring of the specified string. The arguments specify the numbers of the characters in the specified string that begin and end the substring.

```
string string1 = "abcdefgh";
string string2;
initial
begin
string2 = string1.substr(1,5);
$display("string2 = %s",string2);
end
```

The `$display` system task displays the following:

```
string2 = bcdef
```

String Conversion Methods

SystemVerilog has the following methods for converting strings:

atoi() atohex() atooct() and atobin()

Returns the integer corresponding to either the ASCII decimal, hexadecimal, octal, or binary representation of the string.

```
string string1 = "10";
reg [63:0] r1;

initial
begin
$monitor("r1 = %0d at %0t",r1,$time);
#10 r1 = string1.atoi;
#10 r1 = string1.atohex;
#10 r1 = string1.atooct;
#10 r1 = string1.atobin;
```

```
end
```

The `$monitor` system task display the following:

```
r1 = x at 0
r1 = 10 at 10
r1 = 16 at 20
r1 = 8 at 30
r1 = 2 at 40
```

atoreal()

Returns a real number that is the decimal value of a string.

```
module m;

real r1;

string string1 = "1235/x0090";

initial
begin
r1 = string1.atoreal;
$display("r1 = 0%f",r1);
end
endmodule
```

The `$display` system task displays:

```
r1 = 1235.000000
```

itoa()

Stores the ASCII decimal representation of an integer in a string.

```
reg [63:0] r1 = 456;
string string1;

initial
begin
string1.itoa(123);
```

```

if (string1 == "123")
    $display("string1 %s",string1);
string1.itoa(r1);
if (string1 == "456")
    $display("string1 %s",string1);
end

```

The `$display` system tasks display:

```
string1 123
```

hextoa()

`hextoa(arg)` returns the ASCII hexadecimal representation of the arg.

octtoa()

`octtoa(arg)` returns the ASCII octal representation of the arg.

bintoa()

`bintoa(arg)` returns the ASCII binary representation of the arg.

realtoa()

`realtoa(arg)` returns the ASCII real representation of the arg.

The following program explains the usage of these string methods.

```

program test();

string s;
real r;
logic [11:0] h = 'hfal;
reg [11:0] o = 'o172;
bit [5:0] b = 'b101010;

task t1();

```



```

$display("-----Start of Program -----");
s.hextoa(h);
$display("Ascii of hex value 'hfal is %s",s);

s.octtoa(o);
$display("Ascii of octal value 'o172 is %s",s);

s.bintoa(b);
$display("Ascii of binary value 'b101010 is %s",s);

s = "12.3456";
r = s.atoreal;
$display("Real value of ascii string \"12.3456\" is
        %f", r);

s = "";
s.realtoa(r);
$display("Ascii of real value 12.3456 is %s",s);
$display("----- End of Program -----");
endtask

initial
    t1();

endprogram

```

The output of this program is:

```

start of Program -----
Ascii of hex value 'hfal is fal
Ascii of octal value 'o172 is 172
Ascii of binary value 'b101010 is 101010
Real value of ascii string "12.3456" is 12.345600
Ascii of real value 12.3456 is 12.3456
----- End of Program -----

```

Predefined String Methods

SystemVerilog provides several class methods to match patterns within strings.

search()

The `search()` method searches for a pattern in the string and returns the index number to the beginning of the pattern. If the pattern is not found, then the function returns -1. The syntax is:

```
integer string_variable.search(string pattern);
```

Here, the argument must be a string.

The following example illustrates the usage of the `search()` class method.

```
integer i;  
string str = "SystemVerilog supports search( ) method";  
i = str.search("supports");  
printf("%d \n", i);
```

This example assigns the index 14 to integer `i` and prints out 14.

match()

The `match()` method processes a regular expression pattern match. It returns 1 if the pattern is found else, it returns 0 if the pattern is not found. The syntax is:

```
integer string_variable.match(string pattern);
```

Here, the pattern must be a regular Perl expression.

The following example illustrates the usage of the `match()` class method.

```
integer i;
string str;
str = "SystemVerilog supports match( ) method";
i = str.match("mat");
```

This example assigns the value 1 to integer i because the pattern “mat” exists within the string str.

prematch()

The prematch() method returns the string that is located just before the string found by the last match() function call. The syntax is:

```
string string_variable.prematch();
```

The following example illustrates the usage of the prematch() class method.

```
integer i;
string str, str1;
str = "SystemVerilog supports prematch( ) method";
i = str.match("supports");
str1 = str.prematch();
```

This example assigns the value “SystemVerilog” to string str1.

postmatch()

The postmatch() method returns the string that is located just after the string found by the last match() function call. The syntax is:

```
string string_variable.postmatch();
```

The following example illustrates the usage of postmatch() class method.

```
integer i;
```

```
string str, str1;
str = "SystemVerilog supports postmatch( ) method";
i = str.match("postmatch( )");
str1 = str.postmatch();
```

This example assigns the value “method” to string str1.

thismatch()

The thismatch() method returns the matched string, based on the result of the last match() function call. The syntax is:

```
string string_variable.thismatch();
```

The following example illustrates the usage of the thismatch() class method.

```
integer i;
string str, str1;
str = "SystemVerilog supports thismatch( ) method";
i = str.match("thismatch");
str1 = str.thismatch();
```

This example assigns the value “thismatch” to string str1.

backref()

The backref() method returns the matched patterns, based on the last match() function call. The syntax is:

```
function string string_variable.backref(integer index);
```

Here, index is the integer number of the Perl expression being matched. Indexing starts at 0.

This function matches a string with Perl expressions specified in a second string.

The following example illustrates the usage of the `backref()` function call.

```
integer i;
string str, patt, str1, str2;
str = "1234 is a number."
patt = "([0-9]+) ([a-zA-Z .]+)";
i = str.match(patt);
str1 = str.backref(0);
str2 = str.backref(1);
```

This example checks the Perl expressions given by string `patt` with string `str`. It assigns the value "1234" to string `str1` because of the match to the expression "[0-9]+". It assigns the value "is a number." to string `str2` because of the match to the expression "[a-zA-Z .]+". Any number of additional Perl expressions can be listed in the `patt` definition, and then called using sequential index numbers with the `backref()` function.

Program Blocks

A program block contains the testbench for a design. In the default implementation of SystemVerilog testbench constructs, all these constructs must be in one program block. Multiple program blocks is an LCA feature.

Requiring these constructs in a program block help to distinguish between the code that is the testbench and the code that is the design.

Program blocks begin with the keyword `program`, followed by a name for the program, followed by an optional port connection list, followed by a semi colon (;). Program blocks end with the keyword `endprogram`, for example:

```
program prog (input clk, output logic [31:0] data, output
             logic ctrl);
  logic dynamic_array [];
  logic assoc_array[*];
  int intqueue [$] = {1,2,3};

  class classA;
  function void vfunc(input in1, output out1);
  .
  .
  .
  endfunction
  .
  .
  .
  endclass

  semaphore sem1 =new (2);
  mailbox mbx1 = new();

  reg [7:0] reg1;
  covergroup cg1 @(posedge clk);
  cp1: coverpoint reg1;
  :
  endgroup
endprogram

bit clk = 0;
logic [31:0] data;
logic ctrl;

module clkmod;
.
.
```

```
.  
prog prog1 (clk,data,ctrl); // instance of the program  
.br/>.br/>.br/>endmodule
```

In many ways a program definition resembles a module definition and a program instance is similar to a leaf module instance but with special execution semantics.

A program block can contain the following:

- data type declarations including initial values. Dynamic arrays, associative arrays, and queues are implemented for program blocks.
- user-defined tasks and functions
- initial blocks for procedural code (but not always blocks)
- final block (please refer [Final Blocks](#) for more details)
- class definitions
- semaphores
- mailboxes
- concurrent assertions
- coverage groups

When VCS executes all the statements in the initial blocks in a program, simulation comes to an end.

Final Blocks

The final block is Limited Customer availability (LCA) feature in NTB (SV) and requires a separate license. Please contact your Synopsys AC for a license key.

A final block executes in the last simulation time step. The following example contains a final block:

```
`timescale 1ns/1ns
module test;
logic l1, l2;

initial
begin
#10 l1=0;
#10 l1=1;
#10 l1=0;
#10 l1=1;
#10 $finish;
end

always @ (posedge l1)
$display("l1 = %0b at %0t",l1,$time);

final $display(" simulation ends at %0t", $time);
endmodule
```

The `$display` system tasks display the following:

```
l1 = 1 at 20
l1 = 1 at 40
           simulation ends at 50
```

The final block executes in the last simulation time step at time 50.

A final block is the opposite of an initial block in that an initial block begins execution in the first simulation time step and a final block executes in the last simulation time step. Apart from the execution

time there are other important differences in a final block. A final block is like a user-defined function call in that it executes in zero simulation time and cannot contain the following:

- delay specifications
- event controls
- nonblocking assignment statements
- wait statements
- user-defined task enabling statements when the user-defined task contains delay specifications, event controls, wait statements, or nonblocking assignment statements

Multiple Program Support

The Multiple program block is Limited Customer availability (LCA) feature in NTB (SV) and requires a separate license. Please contact your Synopsys AC for a license key.

Multiple programs support enables multiple testbenches to run in parallel. Use this when testbenches model standalone components for example, Verification IP (or work from a previous project). Because components are independent, direct communication between them except through signals is undesirable. For example, a UART and CPU model would communicate only through their respective interfaces, and not through the testbench. Thus, multiple programs modeling standalone components allows usage without having knowledge of the code given, or requiring modifications to your own testbench

Arrays

Dynamic Arrays

Dynamic arrays are unpacked arrays with a size that can be set or changed during simulation. The syntax for a dynamic array is as follows:

```
data_type name [];
```

The empty brackets specify a dynamic array.

The currently supported data types for dynamic arrays are as follows:

<code>bit</code>	<code>logic</code>	<code>reg</code>	<code>byte</code>
<code>int</code>	<code>longint</code>	<code>shortint</code>	<code>integer</code>
<code>time</code>	<code>string</code>	<code>class</code>	<code>enum</code>
<code>events</code>	<code>mailbox</code>	<code>semaphore</code>	

The `new[]` Built-In Function

The `new[]` built-in function is for specifying a new size for a dynamic array and optionally specifying another array whose values are assigned the dynamic array. Its syntax is as follows:

```
array_identifier = new[size] (source_array);
```

The optional (*source_array*) argument specifies another array (dynamic or fixed-size) whose values VCS assigns to the dynamic array. If you don't specify the (*source_array*) argument, VCS initializes the elements of the newly allocated array to their default value.

The optional (*source_array*) argument must have the same data type as the array on the left-hand side, but it need not have the same size. If the size of (*source_array*) is less than the size of the new array, VCS initializes the extra elements to their default values. If the size of (*source_array*) is greater than the size of the new array, VCS ignores the additional elements.

```
program prog;
.
.
.
bit bitDA1 [];
bit bitDA2 [];
bit bitDA3 [];
bit bitSA1 [100];
logic logicDA [];
.
.
.
initial
begin
bitDA1 = new[100];
bitDA2 = new[100] (bitDA2);
bitDA3 = new[100] (bitSA1);
logicDA = new[100];
end
.
.
.
endprogram
```

The size() Method

The `size` method returns the current size of a dynamic array. You can use this method with the `new[]` built-in function, for example:

```
bitDA3 = new[bitDA1.size] (bitDA1);
```

The delete() Method

The `delete` method sets a dynamic array's size to 0 (zero). It clears out the dynamic array.

```
bitDA1 = new[3];
$display("bitDA1 after sizing, now size =
        %0d",bitDA1.size);
bitDA1.delete;
$display("bitDA1 after sizing, now size =
        %0d",bitDA1.size);
```

VCS displays from this code:

```
bitDA1 after sizing, now size = 3
bitDA1 after sizing, now size = 0
```

Assignments to and from Dynamic Arrays

You can assign a dynamic array to and from a fixed-size array, queue, or another dynamic array, provided they are of equivalent data types, for example:

```
logic lFA1[2];
logic lDA1[];
initial
begin
$display("lDA1 size = %0d",lDA1.size);
```

```

lFA1[1]=1;
lFA1[0]=0;
lDA1=lFA1;
$display("lDA1[1] = %0d", lDA1[1]);
$display("lDA1[0] = %0d", lDA1[0]);
$display("lDA1 size = %0d",lDA1.size);
end
endprogram

```

VCS displays:

```

lDA1 size = 0lDA1[1] = 1
lDA1[0] = 0
lDA1 size = 2

```

When you assign a fixed-size array to a dynamic array, the dynamic array's size changes to the size of the fixed-size array. This is also true when you assign a dynamic array with a specified size to another dynamic array, for example:

```

logic lDA1[];
logic lDA2[];
initial
begin
lDA1=new[2];
$display("lDA2 size = %0d",lDA2.size);
lDA1[1]=1;
lDA1[0]=0;
lDA2=lDA1;
$display("lDA2[1] = %0d", lDA2[1]);
$display("lDA2[0] = %0d", lDA2[0]);
$display("lDA2 size = %0d",lDA2.size);
end
endprogram

```

This code displays the following:

```

lDA2 size = 0
lDA2[1] = 1

```

```
lDA2[0] = 0
lDA2 size = 2
```

You can assign a dynamic array to a fixed-size array, provided that they are of equivalent data type and that the current size of the dynamic array matches the size of the fixed-size array.

Associative Arrays

An associative array has a lookup table for the elements of its declared data type. Its index is a data type which serves as the lookup key for the table. This index data type also establishes an order for the elements.

The syntax for declaring an associative array is as follows:

```
data_type array_id [index_type];
data_type array_id [* | string];
```

Where:

data_type

Is the data type of the associative array.

array_id

Is the name of the associative array.

index_type

Specifies the type of index. Only two types are currently implemented. They are as follows:

*

Specifies a wildcard index.

string

Specifies a string index.

Wildcard Indexes

You can enter the wildcard character as the index.

```
data_type array_id [*];
```

Using the wildcard character permits entering any integral data type as the index. Integral data types represent an integer (`shortint`, `int`, `longint`, `byte`, `bit`, `logic`, `reg`, `integer`, and also packed structs, packed unions, and `enum`).

Note:

The wildcard index has a 64 bit limitation.

```
program m;
bit [2:0] AA1[*];
int int1;
logic [7:0] log1;

initial begin
    int1 = 27;
    log1 = 42;
    AA1[456] = 3'b101;
    AA1[int1] = 3'b000;    // index is 27
    AA1[log1] = 3'b111;    // index is 42
end
endprogram
```

String Indexes

A string index specifies that you can index the array with a string. You specify a string index with the keyword `string`.

```
program p;

logic [7:0] a[string];
string string_variable;
```

```
initial begin
    a["sa"] = 8;
    a["bb"] = 15;
    a["ec"] = 29;
    a["d"] = 32;
    a["e"] = 45;
    a[string_variable] = 1;

end
endprogram
```

Associative Array Assignments and Arguments

You can only assign an associative array to another associative array with a equivalent data type. Similarly, you can only pass an associative array as an argument to another associative array with a equivalent data type.

Associative Array Methods

There are methods for analyzing and manipulating associative arrays.

`num`

Returns the number of entries in the array.

`delete`

Removes all entries from an array. If you specify an index, this method removes the entry specified by the index.

`exists`

Returns a 1 if the specified entry exists.

`first`

Assigns the value of the smallest or alphabetically first entry in the array. Returns 0 if the array is empty and returns 1 if the array contains a value.

`last`

Assigns the value of the largest or alphabetically last entry in the array. Returns 0 if the array is empty and returns 1 if the array contains a value.

`next`

Finds the entry whose index is greater than the given index. If there is a next entry, the index variable is assigned the index of the next entry, and the function returns 1. Otherwise, variable is unchanged, and the function returns 0

`prev`

Finds the entry whose index is smaller than the given index. If there is a previous entry, the index variable is assigned the index of the previous entry, and the function returns 1. Otherwise, variable is unchanged, and the function returns 0.

The following example shows how to use these methods.

```
program p;
logic [7:0] a[string];
string s_index;
initial begin
    a["sa"] = 8;
    a["bb"] = 15;
    a["ec"] = 29;
    a["d"] = 32;
    a["e"] = 45;
    $display("number of entries = %0d",a.num);
    if(a.exists("sa"))
        $display("string \"sa\" is in a");
    if(a.first(s_index))
        begin
```

```

        $display("the first entry is
                \"%s\"",s_index);
    do
        $display("%s :
                %0d",s_index,a[s_index]);
    while (a.next(s_index));
end

    if(a.last(s_index))
    begin
        $display("the last entry is
                \"%s\"",s_index);
    do
        $display("%s :
                %0d",s_index,a[s_index]);
    while (a.prev(s_index));
    end
    a.delete;
    $display("number of entries = %0d",a.num);
end
endprogram

```

VCS displays the following:

```

number of entries = 5
string "sa" is in a
the first entry is "bb"
bb : 15
d : 32
e : 45
ec : 29
sa : 8
the last entry is "sa"
sa : 8
ec : 29
e : 45
d : 32
bb : 15
number of entries = 0

```

Queues

A queue is an ordered collection of variables with the same data type. The length of the queue changes during simulation. You can read any variable in the queue, and insert a value anywhere in the queue.

The variables in the queue are its elements. Each element in the queue has a number: 0 is the number of the first, you can specify the last element with the `$` (dollar sign) symbol. The following are some examples of queue declarations:

```
logic logque [$];
```

This is a queue of elements with the `logic` data type.

```
int intque [$] = {1,2,3};
```

This is a queue of elements with the `int` data type. These elements are initialized 1, 2, and 3.

```
string strque [$] = {"first","second","third","fourth"};
```

This is a queue of elements with the `string` data type. These elements are initialized "first", "second", "third", and "fourth".

You assign the elements to a variable using the element number, for example:

```
string s1, s2, s3, s4;
initial
begin
s1=strque[0];
s2=strque[1];
s3=strque[2];
s4=strque[3];
$display("s1=%s s2=%s s3=%s s4=%s",s1,s2,s3,s4);
.
.
.
```

```
end
```

The `$display` system task displays:

```
s1=first s2=second s3=third s4=fourth
```

You also assign values to the elements using the element number, for example:

```
int intque [$] = {1,2,3};
initial
begin
  intque[0]=4;
  intque[1]=5;
  intque[2]=6;
  $display("intque[0]=%0d intque[1]=%0d intque[2]=%0d",
  intque[0],intque[1],intque[2]);
  .
  .
  .
end
```

The `$display` system task displays:

```
intque[0]=4 intque[1]=5 intque[2]=6
```

Concatenation operations, for adding elements, are not yet supported, for example:

```
intque = {0,intque};
intque = {intque, 4};
```

Removing elements from a queue are not yet supported, for example:

```
strque = strque [1:$];
intque = intque[0:$-1];
```

Queue Methods

There are the following built-in methods for queues:

`size`

Returns the size of a queue.

```
program prog;
  int intque [$] = {1,2,3};

  initial
  begin
    for (int i = 0; i < intque.size; i++)
      $display(intque[i]);
  end
endprogram
```

`insert`

Inserts new elements into the queue. This method takes two arguments: the first is the number of the element, the second is the new value.

```
program prog;
  string strque [$] = {"first","second","third","forth"};

  initial
  begin
    for (int i = 0; i < strque.size; i++)
      $write(strque[i], " ");
    $display(" ");
    strque.insert(1, "next");
    strque.insert(2, "somewhere");
    for (int i = 0; i < strque.size; i++)
      $write(strque[i], " ");
    $display(" ");
  end
endprogram
```

The `$display` system tasks display the following:

```
first second third forth
first next somewhere second third forth
```

`delete`

Removes an element from the queue, specified by element number. If you don't specify an element number, this method deletes all elements in the queue.

```
string strque [$] = {"first","second","third"};
```

```
initial
begin
for (int i =0; i<strque.size; i++)
    $write(strque[i], " ");
$display(" ");
strque.delete(1);
for (int i =0; i<strque.size; i++)
    $write(strque[i], " ");
end
```

The system tasks display the following:

```
first second third
first third
```

`pop_front`

Removes and returns the first element of the queue.

```
string strque [$] = {"first","second","third"};
string s1;
initial
begin
$write("the elements of strque are ");
for (int i =0; i<strque.size; i++)
    $write(strque[i], " ");
$display("\ns1 before pop contains %s ",s1);
s1 = strque.pop_front();
$display("s1 after pop contains %s ",s1);
$write("the elements of strque are ");
```

```

for (int i =0; i<strque.size; i++)
    $write(strque[i], " ");
end

```

The system tasks display the following:

```

the elements of strque are first second third
s1 before pop contains
s1 after pop contains first
the elements of strque are second third

```

```

string strque [$] = {"first","second","third"};
string s1;
initial
begin
for (int i =0; i<strque.size; i++)
    $write(strque[i], " ");
$display(" ");
s1 = strque.pop_front;
for (int i =0; i<strque.size; i++)
    $write(strque[i], " ");
end

```

The system tasks display the following:

```

first second third
second third

```

pop_back

Removes and returns the last element in the queue.

```

program prog;
string strque [$] = {"first","second","third"};
string s1;
initial
begin
$write("the elements of strque are ");
for (int i =0; i<strque.size; i++)
    $write(strque[i], " ");
$display("\ns1 before pop contains %s ",s1);
s1 = strque.pop_back;

```

```

$display("s1 after pop contains %s ",s1);
$write("the elements of strque are ");
for (int i =0; i<strque.size; i++)
    $write(strque[i]," ");
end
endprogram

```

The system tasks display the following:

```

the elements of strque are first second third
s1 before pop contains
s1 after pop contains third
the elements of strque are first second

```

push_front and push_back

Add elements to the front and back of the queue.

```

int intque [$] = {1,2,3};

initial
begin
for( int i = 0; i < intque.size; i++)
    $write(intque[i]," ");
intque.push_front(0);
intque.push_back(4);
$write(" \n");
for( int i = 0; i < intque.size; i++)
    $write(intque[i]," ");
end

```

The system tasks display the following:

```

1 2 3
0 1 2 3 4

```

The foreach Loop

A `foreach` loop iterates through an array. Its argument is any kind of array and the loop variables designate the indexes of the array. The following is an elementary example of the use of this construct:


```

module test;
bit [11:10][9:8][7:0] bit_array1;
initial
begin
foreach (bit_array1[dim1,dim2])
    bit_array1 [dim1][dim2]=dim1*dim2;
foreach (bit_array1[dim1, dim2])
    $display("bit_array1[%1d] [%1d]=%0d",
dim1,dim2,bit_array1[dim1][dim2]);
end
endmodule

```

The bit data type array named `bit_array1` has three dimensions. The first `foreach` loop iterates through the first two dimensions 11:10 and 9:8, to assign 8-bit values to these elements.

The second `foreach` loop displays the deposited values.

The `$display` system task displays the following:

```

bit_array1 [11][9]=99
bit_array1 [11][8]=88
bit_array1 [10][9]=90
bit_array1 [10][8]=80

```

The `foreach` loop also works with other types of arrays, such as this example of a string array:

```

module test;
string words [2];
initial
begin
words [1] = "verification";
words [0] = "simulation";
foreach (words [j])
    $display("string element number %1b",j,
"contains \"",words[j], "\"");
end

```

```
endmodule
```

The \$display system task displays the following:

```
string element number 0 contains "simulation"  
string element number 1 contains "verification"
```

The foreach loop also works with queues and dynamic and associative arrays. The following is an example with a dynamic array:

```
program test;  
integer fixed_int_array[3] = {0, 1, 2};  
integer dynamic_int_array[];  
initial  
begin  
dynamic_int_array=new[3](fixed_int_array);  
foreach (dynamic_int_array[dim1])  
    $display("dynamic_int_array [%1d]=%0d",  
            dim1,dynamic_int_array[dim1]);  
end  
endprogram
```

The \$display system task displays the following:

```
dynamic_int_array [0]=0  
dynamic_int_array [1]=1  
dynamic_int_array [2]=2
```

The following is an example with an associative array:

```
program test;  
bit [2:0] assoc_array1 [*];  
initial  
begin  
assoc_array1[0]=3'b000;  
assoc_array1[1]=3'b001;
```

```

assoc_array1[7]=3'b111;
foreach (assoc_array1[dim1])
    $display("assoc_array1 [%1d]=%0d",
            dim1,assoc_array1[dim1]);
end
endprogram

```

The `$display` system task displays the following:

```

assoc_array1 [0]=0
assoc_array1 [1]=1
assoc_array1 [7]=7

```

Array Aggregates (Reduction/Manipulation) Methods in Constraints

SystemVerilog includes a set of array reduction methods which allow declarations of complex constraints for arrays and queues in a compact and flexible format.

The following is the syntax for these methods:

```

function array_or_expression_type method
    (array_type iterator = item)

```

The array aggregate expression is a valid part of a constraint expression and can be used any place that a variable can be used, with the exception of solve-before constraints.

List of Aggregate Methods

Table 24-1

Method	Description
sum()	Performs addition and returns sum of array elements
product()	Performs multiplication, and returns product of array elements
and()	Performs bitwise AND operation, and returns bit-wise AND of array elements.
or()	Performs bitwise OR operation, and returns bit-wise OR of array elements
xor()	Performs logical XOR operation, and returns logical XOR of all array elements.

The following code is an example of the aggregate method, sum() in an inline constraint block:

```
program test;

class myclass;

    rand int b[2:0];
    int a[3:0] = {2,4,6,8};

    function void post_randomize();
        if((b[0] == 28) && (b[2] == 28))
            $display("test Passed");
        else
            $display("test Failed");
    endfunction
endclass

myclass mc = new;
initial begin
    int i;
    i = mc.randomize() with { foreach (b [j]) b[j] ==
        a.sum(temp) with (a[temp.index] + 2);};
end
```

```

        if(i)
            $display("Randomization Success");
        else
            $display("Randomization Fails");
        end
    endprogram

```

Notes

1. Empty array - If the array over which the aggregate operation is defined has size 0, then the behavior is as follows:

- *array.sum()*: The expression is substituted by 0.
- *array.product()*: The expression is substituted by 1.

For all other aggregate methods there is a runtime error if reference to the aggregate operation is in an ON constraint block.

2. Array types - The array aggregate methods and the foreach loop support the fixed size array, dynamic array, associative array, and SmartQ types of arrays in every context.

Unsupported features in Array aggregates:

- Using virtual interfaces to refer to variables inside array constraints.
- Array aggregates of SV style (and, xor, or, etc.)
- Array aggregates outside constraint blocks (and, xor, or, etc).
- Array aggregates with usage as follows: *array.cum* with (*item == item.index*).

Classes

The user-defined data type, class, is composed of data members of valid SystemVerilog data types (known as properties) and tasks or functions (known as methods) for manipulating the data members. The properties and methods, taken together, define the contents and capabilities of a class instance (also referred to as an object).

Use the `class` and `endclass` keywords to declare a class.

```
class B;
    int q = 3;
    function int send (int a);
        send = a * 2;
    endfunction

    task show();
        $display("q = %0d", q);
    endtask
endclass
```

In the above example, the members of class “B” are the property, `q`, and the methods `send()` and `show()`.

Note:

- See “Class Packet Example” on page 24-66 for a more complex example of a class declaration. The name of the class is “Packet.”
- Unpacked unions inside classes are not yet supported. Packed unions and packed and unpacked structures inside classes are supported.

```
class myclass;
typedef struct packed{ int int1;
```

```

        logic [7:0] log1;
        } struct1;

struct1 mystruct;

typedef struct packed { int intt1; logic [31:0] logg1; }
pstruct1;

typedef union packed {
    pstruct1 u_pstruct1;
    } p_union1;

endclass

```

Creating an Instance (object) of a Class

A class declaration is the template from which objects are created. When a class is constructed the object is built using all the properties and methods from the class declaration.

To create an object (that is, an *instance*) of a declared class, there are two steps. First, declare a handle to the class (a handle is a reference to the class instance, or object):

```
class_name handle_name;
```

Then, call the `new()` class method:

```
handle_name = new();
```

The following example expands on the above example to show how to create an instance of class “B.”

```

program P;
    class B;
        int q = 3;
        function int send (int a);

```

```

        send = a * 2;
    endfunction

    task show();
        $display("q = %0d", q);
    endtask
endclass

initial begin
    B b1; //declare handle, b1
    b1 = new; //create an object by calling new
end
endprogram

```

The above two steps can be merged into one for instantiating a class at the time of declaration:

```

class_name handle_name = new();

```

For example:

```

B b1 = new;

```

The `new()` method of the class is a method which is part of every class. It has a default implementation which simply allocates memory for the object and returns a handle to the object.

Constructors

You can declare your own `new()` method. In such a case, the prototype that you must follow is:

```

function new([arguments]);
    // body of method
endfunction

```

Use this syntax when using the user-defined constructor:


```
handle_name = new([arguments]);
```

Arguments are optional.

Below is an example of a user-defined constructor:

```
program P;

class B; //declare class
    integer command;
    function new(integer inCommand = 1);
        command = inCommand;
        $display("command = %d", command);
    endfunction
endclass
endprogram
```

When called, new() will print the value passed to it as the value of command.

When a class property has been initialized in its declaration, the user-defined new() constructor can be used to override the initialized value. The following example is a variant of the first example in this section.

```
program P;
    class A;
        int q = 3; //q is initialized to 3
        function new();
            q = 4; //constructor overrides & assigns 4
        endfunction
    endclass
    A a1;
    initial
        begin
            a1 = new;
            $display("The value of q is %0d.", a1.q);
        end
endprogram
```

The output of this program is:

```
The value of q is 4.
```

If a property is not initialized by the constructor, it is implicitly initialized, just like any other variable, with the default value of its data type.

Assignment, Re-naming and Copying

Consider the following:

```
class Packet;  
    ...  
endclass  
  
Packet p1;
```

`Packet p1;` creates a variable, `p1`, that can hold the handle of an object of class `Packet`. The initial default value of `p1` is null. The object does not yet exist, and `p1` does not contain an actual handle, until an instance of type `Packet` is created as shown below:

```
p1 = new(); //if no arguments are being passed, () can be  
           //omitted. e.g., p1=new;
```

Another variable of type `Packet` can be declared and assigned the handle, `p1` to it as shown below:

```
Packet p2;  
p2 = p1;
```

In this case, there is still only *one* object. This single object can be referred to with either variable, `p1` or `p2`.

Now consider the following, assuming `p1` and `p2` have been declared and `p1` has been instantiated:

```
p2 = new p1;
```

The handle `p2` now points to a *shallow* copy of the object referenced by `p1`. Shallow copying creates a new object consisting of all properties from the source object. Objects are not copied, only their handles. That is, a shallow copy is a duplication of members, including handles, of an object's first level constituents in the hierarchy. However, an object referenced by a copied handle is not, itself, copied. That is, an object contained within an object is not copied, only the referential handle.

Static Properties

The `static` keyword is used to identify a class property that is shared with all instances of the class. A static property is not unique to a single object (that is, all objects of the same type share the property). A static variable is initialized once.

Syntax

```
static data_type variable;
```

Example

```
program test;
  class Parcel;
    static int count = 2;
    function new();
      count++;
    endfunction
  endclass
```

```

        initial begin
            Parcel pk1 = new;
            Parcel pk2 = new;
            Parcel pk3 = new;

            $display("%d Parcel", pk3.count);
            $display("%d Parcel", pk2.count);
            $display("%d Parcel", pk1.count);
        end
    endprogram

```

Output

```

5 Parcel
5 Parcel
5 Parcel

```

In the above example, every time an object of type `Parcel` is created, the `new()` function is invoked and the static variable `count` is incremented. The final value of `count` is “5.” If `count` is declared as non-static, the output of the above program is:

```

3 Parcel
3 Parcel
3 Parcel

```

Global Constant Class Properties

The `const` keyword is used to designate a class property as read-only. Class objects are dynamic objects, therefore either global or instance properties can be designated “`const`.” At present, only global constants are supported.

When declared, a global constant class property includes an initial value. Consider the following example:

```

//global_const.sv
    program test;

```

```

class Xvdata_Class;
    const int pi = 31412;//const variable
endclass

Xvdata_Class data;
    initial begin
        data = new();
        data.pi = 42; //illegal and will generate
                       //compile time error
    end
endprogram

```

The line, `data.pi=42`, is not valid and yields the following compile time error message:

```

Error-[IUCV] Invalid use of 'const' variable
Variable 'pi' declared as 'const' cannot be used
in this context
"global_const.sv", 17: data.pi = 42;

```

Method Declarations: Out of Class Body Declarations

The following example demonstrates the usage of scope resolution for:

- Declaring class function outside class
- Accessing class enum label in inline constraint block
- Declaring constraint block out of class and also accessing class enum label inside that block

```

program prog;

typedef enum {X, Y} A;

class C;
    typedef enum {S1,S2,S3,S4} E; //enum declared inside class

```

```

        rand A a1;
        rand E e1;

        extern function f1();
        constraint c;
    endclass

//out of class body method declared with the help of ::
//operator
function C::f1();
int i;
    $display("Accessing enum label in constraint block
            through :: operator");
    repeat (5)
    begin
        i = this.randomize();
        if(i)
            $display("e1 = %s", this.e1.name);
        else
            $display("Randomization Failed");
    end

    // accessing enum label through :: operator in out
    // of class body method
    $display("Accessing enum label in function block
            through :: operator");
    i = this.randomize with {e1 != C::S2;};
    if(i)
        $display("e1 = %s", this.e1.name);
    else
        $display("Randomization Failed");
endfunction

// out of block constraint declaration accessing enum label
// through :: operator
constraint C::c { a1 == X; e1 inside {C::S1,C::S2,C::S3}; }

C c1 = new;    // creating object of class C

initial begin
    c1.f1();

```

```
        end
    endprogram
```

The Output of this program is

```
Accessing enum label in constraint block through :: operator
e1 = S1
e1 = S2
e1 = S3
e1 = S1
e1 = S1
Accessing enum label in function block through :: operator
e1 = S3
```

Class Extensions

Subclasses and Inheritance

SystemVerilog's OOP implementation provides the capability of inheriting all the properties and methods from a base class, and extending its capabilities within a subclass. This concept is called inheritance. Additional properties and methods can be added to this new subclass.

For example, suppose we want to create a linked list for a class "Packet." One solution would be to extend Packet, creating a new subclass that inherits the members of the parent class (see Class Packet Example on page 66 for class "Packet" declaration).

```
class LinkedPacket extends Packet;
    LinkedPacket next;
    function LinkedPacket get_next();
        get_next = next;
    endfunction
```

```
endclass
```

Now, all of the methods and properties of Packet are part of LinkedPacket - as if they were defined in LinkedPacket - and LinkedPacket has additional properties and methods. We can also override the parent's methods, changing their definitions.

Abstract classes

A group of classes can be created that are all “derived” from a common abstract base class. For example, we might start with a common base class of type BasePacket that sets out the structure of packets but is incomplete; it would never be instantiated. It is “abstract.” From this base class, a number of useful subclasses can be derived: Ethernet packets, token ring packets, GPSS packets, satellite packets. Each of these packets might look very similar, all needing the same set of methods, but they would vary significantly in terms of their internal details. We start by creating the base class that sets out the prototype for these subclasses. Since it will not be instantiated, the base class is made abstract by declaring the class as virtual:

```
virtual class BasePacket;
```

By themselves, abstract classes are not tremendously interesting, but, these classes can also have *virtual* methods. Virtual methods provide prototypes for subroutines, all of the information generally found on the first line of a method declaration: the encapsulation criteria, the type and number of arguments, and the return type if it is needed. When a virtual method is overridden, the prototype must be followed exactly:

```
virtual class BasePacket;  
    virtual function integer send(bit[31:0] data);
```



```

        endfunction
    endclass

    class EtherPacket extends BasePacket;
        function integer send(bit[31:0] data)
            // body of the function
            ...
        endfunction
    endclass

```

`EtherPacket` is now a class that can be instantiated.

If a subclass which is extended from an abstract class is to be instantiated, then all virtual methods defined in the abstract class must have bodies either derived from the abstract class, or provided in the subclass. For example:

```

program P;
virtual class Base;
    virtual task print(string s);
        $display("Base::print() called from %s", s);
    endtask

    extern virtual function string className();

endclass

class Derived extends Base;
    function string className();
        return "Derived";
    endfunction
endclass
Derived d = new;
initial #1 d.print("");
endprogram

```

Methods of non-abstract classes can also be declared virtual. In this case, the method must have a body, since the class, and its subclasses, must be able to be instantiated. It should be noted that once a method is declared as “virtual,” it is forever virtual. Therefore, the keyword does not need to be used in the redefinition of a virtual method in a subclass.

Polymorphism

The ability to call a variety of functions using the exact same interface is called *polymorphism*.

Suppose we have a class called Packet. Packet has a task called display(). All the derived classes of Packet will also have a display() task, but the base version of display() does not satisfy the needs of the derived classes. In such a case we want the derived class to implement its own version of the display() task to be called.

To achieve polymorphism the base class must be abstract (defined using the `virtual` identifier) and the method(s) of the class must be defined as virtual. The abstract class (see page 50 for definition of abstract class) serves as a template for the construction of derived classes.

```
virtual class Packet;  
    extern virtual task display();  
endclass
```

The above code snippet defines an abstract class, Packet. Within Packet the virtual task, display(), has been defined. display() serves as a prototype for all classes derived from Packet. Any class which derives from Packet() must implement the display() task. Note that the prototype is enforced for each derived task (that is, must keep the same arguments, etc.).

```

class MyPacket extends Packet;
    task display();
        $display("This is the display within
                MyPacket");
    endtask
endclass

class YourPacket extends Packet;
    task display();
        $display("This is the display within
                YourPacket");
    endtask
endclass

```

This example illustrates how the two classes derived from Packet implement their own specific version of the display() method.

All derived classes can be referenced by a base class object. When a derived class is referenced by the base class, the base class can access the virtual methods within the derived class through the handle of the base class.

```

program polymorphism;
    //include class declarations of Packet, MyPacket
    //and YourPacket here

    MyPacket mp;
    YourPacket yp;
    Packet p; //abstract base class

    initial
    begin
        mp = new;
        yp = new;
        p = mp; // mp referenced by p
        p.display(); // calls display in MyPacket
        p = yp;
        p.display(); // calls display in YourPacket
    end
endprogram

```

Output:

```
This is the display within MyPacket
This is the display within YourPacket
```

This is a typical, albeit small, example of polymorphism at work.

Scope Resolution Operator ::

Scope resolution operator allows you to access user defined class types as shown in the following example:

```
program p;

    class A;
        typedef enum {red,yellow,blue} color;
        typedef byte MYBYTE;
    endclass

    A::MYBYTE b1 = 8'hff;    //declaring b1 of class byte type
    A::color c1;           //declaring c1 of class enum type

    initial begin
        $display("%h",b1);
        $display("%0d",c1.first);
        $display("%0d",c1.next);
    end
endprogram
```

The Output of the program is:

```
ff
0
1
```

super keyword

The `super` keyword is used from within a derived class to refer to properties of the parent class. It is necessary to use `super` when the property of the derived class has been overridden, and cannot be accessed directly.

```
program sample;
    class Base;
        integer p;
        virtual task display();
            $display("\nBase: p=%0d", p);
        endtask
    endclass

    class Extended extends Base;
        integer q;
        task display();
            super.display(); //refer to Base "display()"
            $display("\nExtended: q=%0d\n", q);
        endtask
    endclass

    Base b1;
    Extended e1;

    initial begin
        b1 = new; // b1 points to instantiated object of Base
        e1 = new; // e1 points to object of Extended
        b1.p = 1; //property "p" of Base initialized to 1
        b1.display(); //will print out "Base: p=1"
        e1.p = 2; // "p" of Base is now 2
        e1.q = 3; // "q" of Extended initialized to 3
        e1.display(); //prints "Base: p=2 Extended: q=3"
    end
endprogram
```

Output of the above program is:

```
Base: p=1
```

```
Base: p=2
```

```
Extended: q=3
```

In the above example, the method, `display()` defined in `Extended` calls `display()`, which is defined in the super class, `Base`. This is achieved by using the `super` keyword.

The property may be a member declared a level up or a member inherited by the class one level up. Only one level of `super` is supported.

When using the `super` keyword within the constructor, `new`, it must be the first statement executed in the constructor. This is to ensure the initialization of the superclass prior to the initialization of the subclass.

Casting

`$cast` enables you to assign values to variables that might not ordinarily be valid because of differing data type.

The following example involves casting a base class handle (“b2”), which actually holds an extended class-handle (“e1”), at runtime back into an extended class-handle (“e2”).

```
program sample;
class Base;
    integer p;
    virtual task display();
        $write("\nBase: p=%0d\n", p);
    endtask
endclass

class Extended extends Base;
```

```

        integer q;
        virtual task display();
            super.display();
            $write("Extended: q=%0d\n", q);
        endtask
    endclass

    Base b1 = new(), b2;
    Extended e1 = new(), e2;

    initial begin
        b1.p = 1;
        $write("Shows the original base property p\n");
        b1.display(); // Just shows base property
        e1.p = 2;
        e1.q = 3;
        $write("Shows the new value of the base property, p,
            and value of the extended property, q\n");

        e1.display(); // Shows base and extended properties
        // Have the base handle b2 point to the extended object
        b2 = e1;
        $write("The base handle b2 is pointing to the
            Extended object\n");
        b2.display(); // Calls Extended.display

        // Try to assign extended object in b2 to extended handle e2
        $write("Using $cast to assign b2 to e2\n");
        if ($cast(e2, b2))
            e2.display(); // Calls Extended.display
        else
            $write("cast_assign of b2 to e2
                failed\n");
        end
    endprogram

```

Output of the above program is:

```

Shows the base property p, which is originally 1
Base: p=1

```

Shows new value of the base property, *p*, and value of the extended property, *q*

Base: *p*=2

Extended: *q*=3

The base handle *b2* is pointing to the Extended object

Base: *p*=2

Extended: *q*=3

Using `$cast` to assign *b2* to *e2*

Base: *p*=2

Extended: *q*=3

The `$cast()` can be used to cast non-enum types into an enum-type destination.

```
$cast(enum1, int1)
```

The following case is supported as long as *enum1* and *enum2* are of same base enum-type:

```
$cast(enum1, enum2)
```

Enumerated types can, however, be cast into any other non-enum types (that is, they can be used as the source argument in `$cast()` without any restrictions):

```
$cast(int1, enum1) // supported, destination non-enum  
    (int)
```

Chaining Constructors

To understand the relational use of the terms “base” and “super” as employed to refer to classes, consider the following code fragment:


```
class ClassA;
    ...
endclass

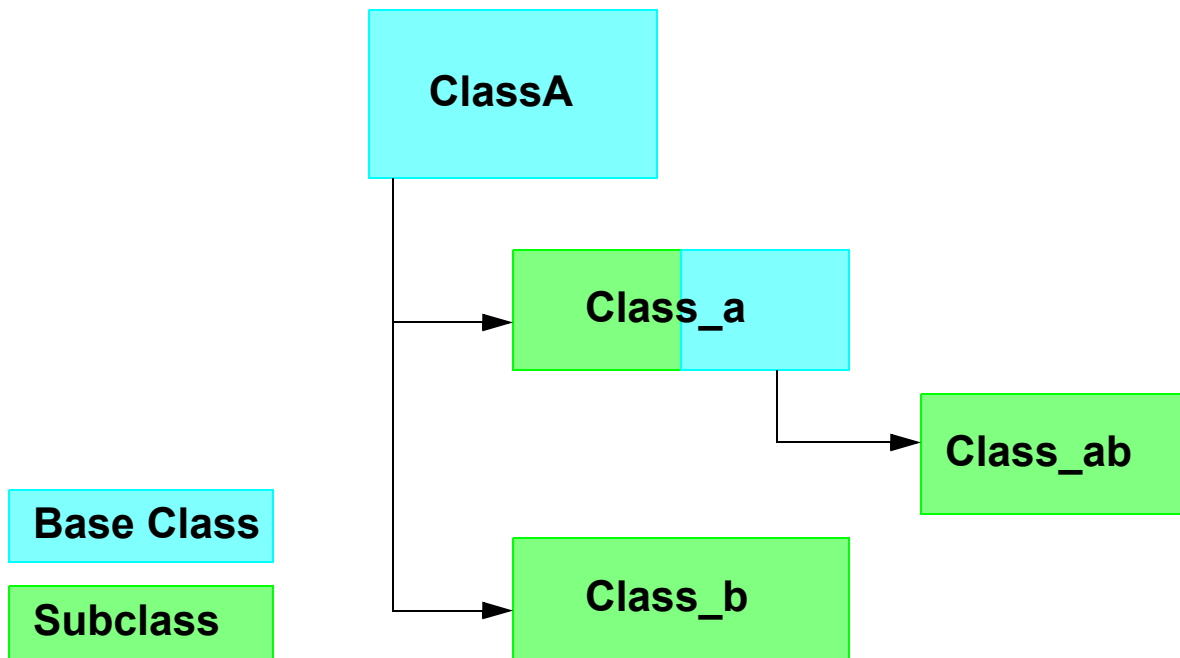
class Class_a extends ClassA;
    ...
endclass

class class_ab extends Class_a;
    ...
endclass

class Class_b extends ClassA;
    ...
endclass
```

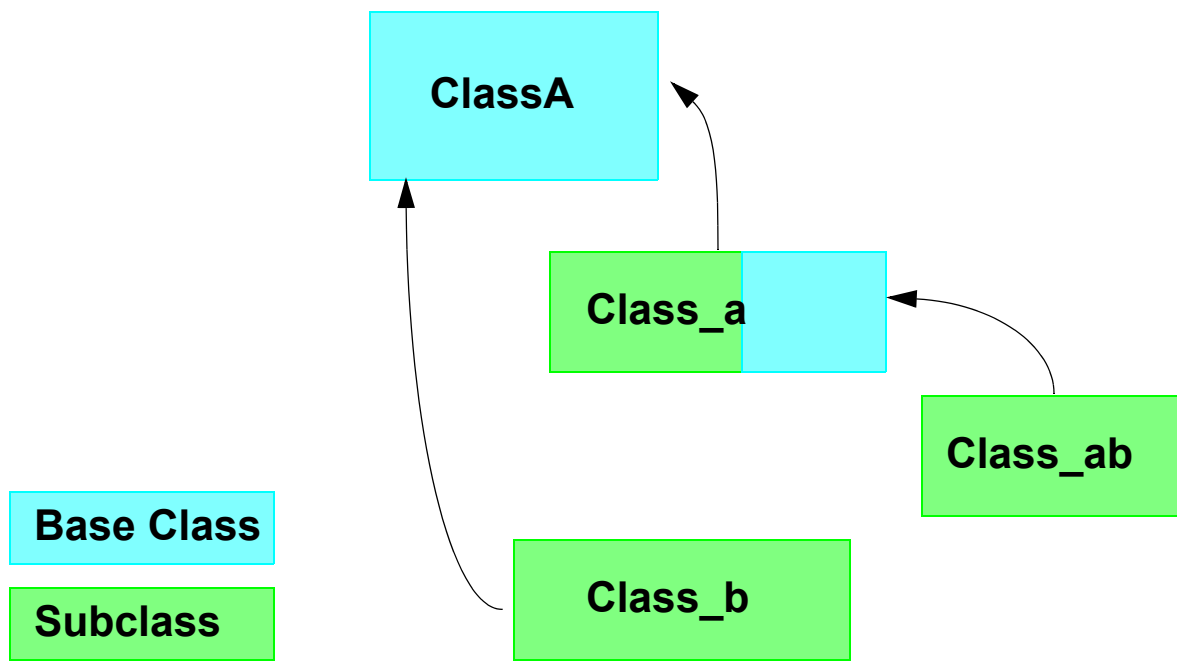
Both `Class_a` and `Class_b` are extended from `ClassA` using the `extends` keyword, making `ClassA` the base class for these two subclasses. `Class_ab` extends from `Class_a`, making `Class_a` the base class for the subclass, `Class_ab`. Both `ClassA` and `Class_a` are super classes since they each have subclasses extended from them.

Figure 24-1 Base and Sub Classes



When a subclass is instantiated, the constructor of the extended super class is implicitly called. If that class, in turn, has a super class, it will implicitly call the constructor of that class. This will continue up the hierarchy until no further super classes exist. In Figure 24-2,

Figure 24-2 Chaining Constructors



when Class_ab is instantiated, the “new” constructor for Class_a is called. In turn, the “new” constructor for ClassA is called. Since ClassA is the last super class in the chain, an object of Class_ab is then created. When Class_b is instantiated, the “new” constructor for ClassA is called and an object of Class_b is created.

This process is called “chaining constructors.”

If the initialization method of the super-class requires arguments, you have two choices. If you want to always supply the same arguments, you can specify them at the time you extend the class:

```
class EtherPacket extends Packet(5);
```

This will pass 5 to the `new()` routine associated with “Packet”.

A more general approach is to use the `super` keyword to call the superclass constructor.

```
function new();  
    super.new(5);  
endfunction
```

If included, the call to `super()` must be the first executable statement (that is, not a declaration) in the constructor.

Accessing Class Members

Properties

A property declaration may be preceded by one of these keywords:

- `local`
- `protected`

The default protection level for class members is “public.” Global access is permitted via `class_name.member`.

In contrast, a member designated as `local` is one that is only visible from within the class itself.

A `protected` class property (or method) has all of the characteristics of a local member, except that it can be inherited, and is visible to subclasses.

Accessing Properties

A property of an object can be accessed using the dot operator (.). The handle name for the object precedes the dot, followed by the qualifying property name (for example, address, command).

instance_name.property_name

Methods

Tasks or functions, known as “methods,” can be designated as local, public, or protected. They are public by default.

Accessing Object Methods

An object’s methods can be accessed using the dot operator (.). The handle for the object precedes the dot, followed by the method.

```
program access_object_method;
    class B;
        int q = 3;
        function int send (int a);
            send = a * 2;
        endfunction

        task show();
            $display("q = %0d", q);
        endtask
    endclass

    initial begin
        B b1;          //declare handle
        b1 = new;     // instantiate
        b1.show();    //access show() of object b.1
        $display("value returned by b1.send() = %0d",
            b1.send(4)); //access send() of object b.1
    end
endprogram
```

Output of program

```
q = 3  
value returned by b1.send() = 8
```

“this” keyword

The `this` keyword is used to unambiguously refer to properties or methods of the current instance. For example, the following declaration is a common way to write an initialization task:

```
program P;  
  class Demo;  
    integer x;  
    task new (integer x);  
      this.x = x;  
    endtask  
  endclass  
endprogram
```

The `x` is now both a property of the class and an argument to the task `new()`. In the task `new()`, an unqualified reference to `x` will be resolved by looking at the innermost scope, in this case the subroutine argument declaration. To access the instance property, we qualify it with `this` to refer to the current instance.

The following is another, complete, example.

```
program P;
  class Demo;
    integer x=4;
    function integer send(integer x);
      this.x = x*3;
    endfunction
    task show();
      $display("The value of x in the object of
        type Demo is = %d", send(this.x));
    endtask
  endclass
  intial begin
    integer x=5;
    Demo D =new;

    D.show();
    $display("The value of the global variable x is
      %0d", x);
  end
endprogram
```

The output of this program is:

```
The value of x in the object of type Demo is = 12
The value of the global variable x is =5
```

Class Packet Example

```
class Packet;
    bit [3:0] command; //property declarations
    bit [40:0] address;
    bit [4:0] master_id;
    integer time_requested;
    integer time_issued;
    integer status;

    function new(); // initialization
        command = 0;
        address = 41'b0;
        master_id = 5'bx;
    endfunction

    task clean(); //method declaration
        command = 0;
        address = 0;
        master_id = 5'bx;
    endtask

    task issue_request(int delay); //method
        // declaration send request to bus
    endtask

    function integer current_status(); //method
        // declaration
        current_status = status;
    endfunction
endclass
```

Unpacked Structures in Classes

The following code examples shows an unpacked structure in a class:

```
program test;
```



```
class myclass;
.
.
.
typedef struct { int int1;
                logic [7:0] log1;
                } struct1;
struct1 mystruct1;
.
.
.
endclass
endprogram
```

Random Constraints

SystemVerilog has constructs for the random testing of a design and a means of constraining the random testing to find hard to reach corner cases.

Note:

To enable the new constraint features, use the `-ntb_opts new_constraints` compile-time option.

Random Variables

You need variables to which VCS assigns random values. There are two types of random variables and two different keywords to begin their declarations:

`rand`

Specifies a standard random variable

`randc`

Specifies a random-cyclic variable.

The following is an example of a standard random variable declaration:

```
rand bit [7:0] randbit1;
```

Variable `randbit1` has eight bits so its possible values range from 0 to 255. The chances that VCS assigns any of these values is precisely 1/256.

The following is an example of a random-cyclic variable declaration:

```
randc bit [1:0] randc1;
```

Variable `randc1` has two bits so its possible values range from 3 to 0. VCS derives a random order, or permutation, of all the possible values, and assigns the values in the permutation in a sequence.

If VCS assigns a value of 3, it won't assign 3 again until it first assigns 2, 1, and 0 (in no particular order), and after it assigns the permutation, it derives the next permutation, beginning with any of the possible values.

If this were a standard random variable, after VCS assigns the 3 value, there is a 1/4 chance that VCS assigns the 3 again, but because it is a random-cyclic variable, after VCS assigns the 3, there is no chance that 3 will also be the next assigned value.

Random variables can only be declared in a class.

```
class Bus;
    rand bit [15:0] addr;
    rand bit [15:0] data;
    .
    .
    .
endclass
```

Constraint Blocks

Constraints are specified in a constraint block. A constraint block can be declared in the same class in which the random variables are declared as well as in a class extended from the base class in which the random variable is defined (see page 58 for definition of “base class”) A constraint block begins with the keyword `constraint`.

```
program prog;
```

```

class myclass;
    rand logic [1:0] log1,log2,log3;
    randc logic [1:0] log4;
    constraint c1 {log1 > 0;}
    constraint c2 {log2 < 3;}
endclass

myclass mc = new;

initial
repeat (10)
    if(mc.randomize()==1)
        $display("log1 = %0d log2=%0d log3=%0d
log4=%0d",mc.log1, mc.log2,
mc.log3, mc.log4);
endprogram

```

In this program block class myclass contains the following:

- Declarations for standard random variables, with the `logic` data type and two bits, named `log1`, `log2`, and `log3`.
- A declaration for cyclic-random variable, with the `logic` data type and two bits, named `log4`.
- Constraint `c1` which says that the random values of `log1` must be greater than 0.
- Constraint `c2` which says that the random values of `log2` must be less than 3.

The `randomize()` method is described in “Randomize Methods” on page 24-100.

The `$display` system task displays the following:

```
log1 = 1 log2=1 log3=1 log4=2
```

```

log1 = 1 log2=2 log3=0 log4=0
log1 = 2 log2=0 log3=1 log4=1
log1 = 1 log2=1 log3=1 log4=3
log1 = 3 log2=2 log3=1 log4=0
log1 = 3 log2=2 log3=3 log4=3
log1 = 3 log2=1 log3=0 log4=2
log1 = 1 log2=1 log3=1 log4=1
log1 = 2 log2=0 log3=2 log4=3
log1 = 1 log2=0 log3=2 log4=0

```

The possible values of all the random variables range from 3 to 0, but the values of log1 are never 0, and the values of log2 are never greater than 3. VCS can assign the same value to log3 over again, but never assigns the same value to log4 over again until it has cycled through all the other legal values.

By means of in-line constraints, a constraint block can be declared for random variables declared in a class while calling the randomize() method on that class' object. (See section entitled "In-line Constraints" on page 108).

```

program test;
  class base;
    rand int r_a;
  endclass
  base b = new;

  initial begin
    int ret;
    ret = b.randomize with {r_a > 0; r_a <= 10;} ;
    // where my declaration {r_a >0 ; r_a <= 10; }
    // is now a constraint block
    if(ret == 1)
      $display("Randomization success");
    else
      $display("Randomization Failed");
    end
  endprogram

```

External Declaration

You can declare a constraint block outside of a class declaration.

```
program prog1;

class c1;
    rand integer rint1, rint2, rint3;
    constraint constr1;
endclass

constraint c1::constr1 { rint1 < rint2; rint2 < rint3; }
endprogram
```

Inheritance

Constraints follow the same rules of inheritance as class variables, tasks, and functions.

```
class myclass;
    rand logic [1:0] log1,log2,log3;
    randc logic [1:0] log4;
    constraint c1 {log1 > 0;}
    constraint c2 {log2 < 3;}
endclass

class myclass2 extends myclass;
    constraint c2 {log2 < 2;}
endclass

myclass2 mc = new;
```

The keyword `extends` specifies that class `myclass2` inherits from class `myclass` and then can change constraint `c2`, specifying that it must be less than 2, instead of less than 3.

Set Membership

You can use the `inside` operator to specify a set of possible values that VCS randomly applies to the random variable.

```
program prog;

class myclass;
    rand int int1;
    constraint c1 {int1 inside {1, [5:7], [105:107]};}
endclass

myclass mc = new;
    initial
    repeat (10)
    if(mc.randomize()==1)
        $display("int1=%0d",mc.int1);
endprogram
```

Constraint `c1` specifies that the possible values for `int1` are as follows:

- 1
- a range beginning with 5 and ending with 7
- a range beginning with 105 and ending with 107

The `$display` system task displays the following:

```
int1=1
int1=7
int1=5
int1=107
int1=106
int1=5
int1=5
int1=6
int1=107
int1=1
```

All these values are equally probable.

In the following example, the `inside` operator is used to specify possible values in an integer array:

```
program test;

class pkt;
    integer bills[0:8] = { 1, 2, 5, 10, 20, 50, 100, 500,
                          1000 };
    rand integer v;
    constraint c4 { v inside bills; }
endclass

int result;
pkt Pkt=new();

initial
    begin
        repeat (10) begin
            result = Pkt.randomize();
            $display("v is %d\n",Pkt.v);
        end
    end
end
endprogram
```

The `$display` system task displays the following:

```
v is          1
v is         100
v is        1000
v is         500
v is          10
v is           1
v is          20
v is           1
```


Weighted Distribution

You can use the `dist` operator to specify a set of values or ranges of values, where you give each value or range of values a weight, and the higher the weight, the more likely VCS is to assign that value to the random variable.

If we modify the previous example as follows:

```
program prog;

class myclass;
    rand int int1;
    constraint c1 {int1 dist {[1:2] := 1, [6:7] := 5, 9
:=10}};
endclass

myclass mc = new;
    int stats [1:9];
    int i;
initial begin
    for(i = 1; i < 10; i++) stats[i] = 0;
    repeat (1700)
        if(mc.randomize()==1) begin
            stats[mc.int1]++;
            //$display("int1=%0d",mc.int1);
        end
    for(i = 1; i < 10; i++)
        $display("stats[%d] = %d", i, stats[i]);
end
endprogram
```

constraint c1 in class myclass is now:

```
constraint c1 {int1 dist {[1:2] := 1, [6:7] := 5, 9
:=10}};
```

Constraint C1 now specifies the possible values of int1 are as follows:

- A range of 1 to 2, each value in this range has a weight of 1. The := operator, when applied to a range, specifies applying the weight to each value in the range.
- A range of 6 to 7 with a weight of 5. The :/ operator specifies applying the weight to the range as a whole.
- 9 with a weight of 10, which is twice as much as the weight for 6.

The repeat loop repeats 1700 times so that we have a large enough sample. The following table shows the various values of int1 during simulation:

value of int	number of times int has this value	fraction of the simulation time
1	105	1/17
2	98	1/17
3	0	
4	0	
5	0	
6	248	2.5/17
7	251	2.5/17
8	0	
9	998	10/17

Implications

An implication is an expression that must be true before VCS applies the constraint.

```

program prog;

class Bus;
    randc bit randcvar;
    bit norandvar;

```

```

        constraint c1 { (norandvar == 1) -> (randcvar ==
            0);}
endclass

Bus bus = new;

initial
    begin
        bus.norandvar = 0;
        #5 bus.norandvar = 1;
        #5 bus.norandvar = 0;
        #5 $finish;
    end

initial
repeat (15)
    #1 if (bus.randomize() ==1)
        $display("randcvar = %0b norandvar = %0b at %0t",
            bus.randcvar, bus.norandvar,$time);
endprogram

```

In constraint c1, when variable norandvar has a 1 value (the implication expression), VCS constrains random variable randcvar to 0. The -> token is the implication operator.

The \$display system task displays the following:

```

randcvar = 0 norandvar = 0 at 1
randcvar = 1 norandvar = 0 at 2
randcvar = 0 norandvar = 0 at 3
randcvar = 1 norandvar = 0 at 4
randcvar = 0 norandvar = 1 at 5
randcvar = 0 norandvar = 1 at 6
randcvar = 0 norandvar = 1 at 7
randcvar = 0 norandvar = 1 at 8
randcvar = 0 norandvar = 1 at 9
randcvar = 1 norandvar = 0 at 10
randcvar = 0 norandvar = 0 at 11
randcvar = 0 norandvar = 0 at 12

```

```
randcvar = 0 norandvar = 0 at 13
randcvar = 1 norandvar = 0 at 14
```

When variable `norandvar` is 0, random-cyclic variable `randcvar` is either 0 or 1. When variable `norandvar` is 1, random-cyclic variable `randcvar` is always 0.

if else Constraints

An alternative to an implication constraint is the `if else` constraint. The `if else` constraint allows a constraint or constraint set on the `else` condition.

```
program prog;

class Bus;
    randc bit [2:0] randcvar;
    bit norandvar;

    constraint c1 { if (norandvar == 1)
                    randcvar == 0;
                    else
                    randcvar inside {[1:3]};}
endclass

Bus bus = new;

initial
begin
    bus.norandvar = 0;
    #5 bus.norandvar = 1;
    #5 bus.norandvar = 0;
    #5 $finish;
end

initial
repeat (15)
    #1 if (bus.randomize() ==1)
```

```

        $display("randcvar = %0d norandvar = %0b at %0t",
                bus.randcvar, bus.norandvar,$time);
endprogram

constraint c1 { if (norandvar == 1)
    randcvar == 0;
    else
        randcvar inside {[1:3]};}

```

Constraint c1 specifies that when variable norandvar has the 1 value, VCS constrains random-cyclic variable randcvar to 0, and when norandvar doesn't have the 1 value, in other words when it has the 0 value, VCS constrains random-cyclic variable randcvar to 1, 2, or 3.

The `$display` system task displays the following:

```

randcvar = 1 norandvar = 0 at 1
randcvar = 2 norandvar = 0 at 2
randcvar = 3 norandvar = 0 at 3
randcvar = 3 norandvar = 0 at 4
randcvar = 0 norandvar = 1 at 5
randcvar = 0 norandvar = 1 at 6
randcvar = 0 norandvar = 1 at 7
randcvar = 0 norandvar = 1 at 8
randcvar = 0 norandvar = 1 at 9
randcvar = 1 norandvar = 0 at 10
randcvar = 2 norandvar = 0 at 11
randcvar = 3 norandvar = 0 at 12
randcvar = 3 norandvar = 0 at 13
randcvar = 2 norandvar = 0 at 14

```

When norandvar is 1, VCS always assigns 0 to randcvar.

Global Constraints

Constraint expressions involving random variables from other objects are called global constraints.

```

program prog;

class myclass;
    rand bit [7:0] randvar;
endclass

class myextclass extends myclass;
    rand myclass left = new;
    rand myclass right = new;
    constraint c1 {left.randvar <= randvar;
                  right.randvar <= randvar;}
endclass
endprogram

```

In this example, class `myextclass` extends class `myclass`. In class `myextclass` are two randomized objects (or instances) of class `myclass`. They are named `left` and `right` and are randomized because they are declared with the `rand` keyword.

Constraint `C1` specifies that the version of `randvar` in the left object must be less than or equal to `randvar` in `myclass`. Similarly the version of `randvar` in the right object must be less than or equal to `randvar` in `myclass`.

Constraint `c1` is a global constraint because it refers to a variable in `myclass`.

Default Constraints

You can specify default constraints by placing the keyword “default” ahead of a constraint block definition, as illustrated in the following example:.

```
[default] constraint constraint_name {constraint_expressions}
```

A default constraint for a particular variable is deemed applicable if no other non-default constraints apply to that variable. The solver satisfy all applicable default constraints for all variables. If the applicable default constraints cannot be satisfied, the solver generates a solver failure.

The following example illustrates the specification of default constraints.

Example 24-1 Default Constraint

```
default constraint foo{
    x > 0;
    x < 5;
}
```

If no other non-default constraints apply to variable x, then the solver satisfies the specified default constraints.

Properties of Default Constraints.

- All constraint expressions in a default constraint block are considered default constraints.
- You can specify multiple default constraints, possibly in multiple constraint blocks, specified for multiple random variables are solved together.
- You can query the status of a default constraint block using `constraint_mode()`, and they can turned ON or OFF.
- You cannot define unnamed constraint blocks (that is, `randomize()` with) as default. The compiler generates an error.
- You can define a default constraint block to any variable visible in the scope where it is declared.

- A default constraint block can be defined externally (that is, in a different file from the one in which the constraint block is declared).
- You can use ordering constraints in default blocks, but these constraints will be treated as non-default constraints.

Overriding default constraints.

Default constraints are applied when they are not overridden by any non-default constraints and contain at least one default variable. Default variables are variables that are `rand` with `rand mode ON` and part of the default constraint. VCS ignores default constraints that do not have any default variables in it.

A non-default constraint must possess the following properties to override a default constraint:

- The constraint mode for the constraint block having a non-default constraint must be ON, or the non-default constraint must be inside a randomize-with constraint block.
- Must have at least one default variable of the default constraint block.
- If the default variable is on the right-hand side of the guarded constraint, the guard must be TRUE, irrespective of it having random variables.
- Must not be a ordering constraint.

The non-default constraint does not override the default constraint if the default variable is in the guard of the non-default constraint.

A default constraint does not override other default constraints under any condition. Thus, if you declare all constraint blocks as default, then none of them will be overridden.

```
program P;

    class C;
        rand reg[3:0] x;
        default constraint c1 {x == 0;}
    endclass
    class D extends C;
        constraint d1 {x > 5;}
    endclass

    initial
    begin
        D d = new();
        d.randomize();
        d.d1.constraint_mode(0);
        d.randomize();
    end
endprogram
```

In the example, the default constraint in block c1 is overridden by the non-default constraint in block d1. The first call to randomize picks a value between 6 and 15 for x, since the non-default constraint is applicable. The default constraint is ignored. The second call switches off the non-default constraint, and a value of 0 will be picked for x in the call to randomize because the default constraint is applied.

```
program P;
    class D;
        rand reg[3:0] x;
        reg y;
        default constraint c1 {x >= 3; x <= 5;}
        constraint d1 {y -> x > 5;}
    endclass
```

```

        initial
        begin
            D d = new();
            d.y = 1;
            d.randomize();
            d.y = 0;
            d.randomize();
        end
    endprogram

```

In the example, the default constraint in block c1 is overridden by the non-default constraint in block d1. The first call to randomize will pick a value between 6 and 15 for x, since the non-default constraint is applicable, given that the guard evaluates to TRUE. The default constraint is ignored. The guard for the non-default constraint evaluates to FALSE in the second call, and a value of between 3 and 5 will be picked for x in the next call to randomize. In this randomize() call, the default constraint is applied.

```

program test;
    class A;
        rand reg[3:0] x;
        rand reg[3:0] y;

        default constraint T{
            x == 10 ; y == 10;
        }
        constraint E {
            y == 0;
        }
        function void post_randomize();
            $display("x = %0d y = %0d\n", x, y);
        endfunction
    endclass

    initial
    begin
        integer ret;
        A a = new;
        ret = a.randomize();
    end
endprogram

```

Output

```
x = 10 y = 0
```

In the example, $x = 10$ since only the second constraint in the default constraint block is overridden.

```
program test;

    class A;
        rand reg[3:0] x;
        rand reg[3:0] y;
        rand reg g;
        default constraint T{x + y == 10;}
        constraint E {g -> y == 1;}

        function void post_randomize();
            $display("x = %0d y = %0d g = %0d\n", x, y, g);
        endfunction
    endclass

    initial
    begin

        integer ret;
        A a = new;

        $display("First call to randomize,
                the guard g evaluates to FALSE\n");
        ret = a.randomize() with {a.g == 0;};

        $display("Second call to randomize, the
                guard g evaluates to TRUE\n");
        ret = a.randomize() with {a.g == 1;};

        $display("Third call to randomize, constraint mode of
                constraint block E is OFF \n");
        a.E.constraint_mode(0);
        a.g.rand_mode(0);
        ret = a.randomize();
    end
endprogram
```

Output:

First call to randomize, the guard g evaluates to FALSE
x = 9 y = 1 g = 0
Second call to randomize, the guard g evaluates to TRUE
x = 3 y = 1 g = 1
Third call to randomize, constraint mode of constraint block E is OFF
x = 1 y = 9 g = 1

In the first call to randomize, the guard is evaluated to FALSE. Therefore, the constraint does not override the default constraint and consequently $x + y == 10$. In the second call to randomize, the guard is evaluated to TRUE. Therefore, the non-default constraint is valid and it overrides the default constraint. In the third call to randomize, the constraint block is turned OFF. Since the non-default constraint is turned off, the default constraint is not overridden and $x + y == 10$.

```
program test;
  class A;
    rand reg[3:0] x;
    rand reg[3:0] y;
    default constraint T{x + y == 10;}
    function void post_randomize();
      $display("x = %0d y = %0d\n", x, y);
    endfunction
  endclass

  initial
  begin
    integer ret;
    A a = new;
    $display("First call to randomize, rand mode of x
              and y are ON\n");
    ret = a.randomize();
    $display("Second call to randomize, rand mode of x
              and y are OFF\n");

    a.x.rand_mode(0);
    a.y.rand_mode(0);
    a.x = 0;
    a.y = 0;
    ret = a.randomize();
    $display("Return status of randomize is %0d" ret);
  end
endprogram
```

Output

```
First call to randomize, rand mode of x and y are ON
x = 3 y = 7
Second call to randomize, rand mode of x and y are OFF
x = 0 y = 0
Return status of randomize is 1
```

In the first call to randomize, the default is not overridden and consequently, `x + y == 10`;

In the second call to randomize, the rand mode of `x` and `y` are turned OFF and both are set to 0. In this call the default variables are rand mode OFF and the default constraint is there fore ignored. If not the `randomize()` would fail and return 0.

Variable Ordering

In an implication like the following:

```
constraint c1 { select -> data == 0;}
```

The select variable “implies” that data is 0, in this case, when `select` is 1, `data` is 0.

The default behavior is for the constraint solver to solve for both `select` and `data` at the same time. Sometimes you can assist the constraint solver by telling it to solve for one variable first. You can do this with another constraint block that specifies the order in which the constraint solver solves the variables:

```
constraint c1 { select -> data == 0;}
constraint c2 ( solve select before data;}
```

Note:

Ordering also changes the distribution of values.

Unidirectional Constraints

The current extension to the constraint language allows the specification of partition points in the constraint set using the unary function `$void()` or using the `hard` keyword in conjunction with `solve-before`, improving the performance and capacity of the solver system.

- `$void()`
- `solve-before hard`

`$void()`. The constraint language allows the use of built-in unary identity function, `$void()`. Any valid constraint expression can be a parameter to the `$void()` function.

Syntax

```
function return_type $void(expression);
```

return_type

Is the same as that of *expression*.

expression

Can be any valid constraint expression.

Example 24-2

```
constraint b1{
    y inside {2,3};
    x % $void(y) == 0;
}
```

Semantics

The `$void()` function imposes an ordering among the variables. All parameters to `$void()` function calls in a constraint are solved for before the constraint is solved.

The support set of a constraint (that is, the set of variables that participate in that constraint) is split into two sets, namely, those that are `$void()` parameters, and those that are not `$void()` parameters.

1. The constraint is not considered for solving when the function parameters incident on it are solved, that is, the function parameters are solved before the constraint is solved.
2. The constraint can be solved when any of the non-function parameters incident on it are solved.
3. The constraint is solved when at least one non-function parameter incident on it is solved, unless the constraint has no non-function parameters incident on it. In that case, the constraint acts as a checker.
4. When the constraint is solved, all previously unsolved non-function parameters incident on it get solved.
5. The ordering directives derived from the use of `void` in constraints in OFF constraint blocks are ignored, and do not affect partitioning in any way.

Consider the following constraint block in example:

```
constraint b1{
    y inside {2,3};
    x % $void(y) == 0;
    x != (y+r);
    z == $void(x*y)
    r == 5;
```

```

}

```

Table 24-2

Constraint	Function Parameter	Non-Function Parameter
<code>y inside {2,3};</code>		<code>y</code>
<code>x%void(y)==0;</code>	<code>y</code>	<code>x</code>
<code>x!=(y+r);</code>		<code>x,y,r</code>
<code>z==void(x*y);</code>	<code>x,y</code>	<code>z</code>
<code>r == 5;</code>		<code>r</code>

Figure 24-1

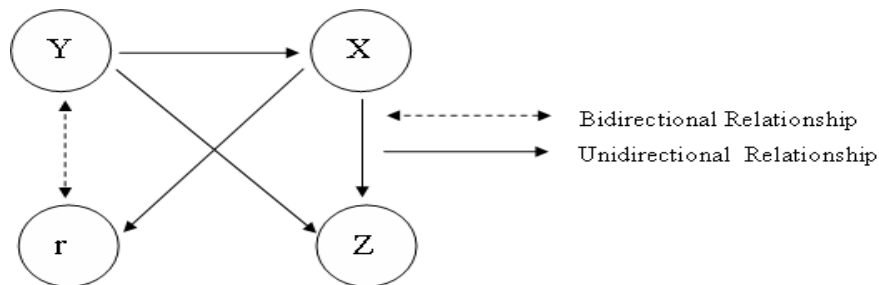


Figure 24-1 illustrates the dependencies between the variables. The variable `y` must be solved before `x` (see `x% $void(y) ==0;`). `y` and `x` must be solved before `z` (see `z==$void(x*y);`). Therefore, the solving order is `y`, `x` and then `z`.

When can `r` be solved for?

- Can `r` be solved at the same time as `y`?

Consider the constraint `x != (y+r)`. If no solving order is specified by unidirectional constraints, then `r` can be solved at the same time as `y` and `x`. However, because of the existence of `x% $void(y) ==0` in the same constraint block, an ordering is imposed. `y`, which is a function parameter in `x%$void(y) ==0`, must be solved before `x`. This means that the three variables cannot be solved for at the same time in `x!=(y+r)`. `y` is solved in `y inside {2,3}`, before `x` and `r` are solved.

- Can r be solved at the same time as x ?

Consider $x \neq (y+r)$ again. Figure 24-1 illustrates that the relationship between r and x is bidirectional which means there is no unidirectional constraint imposed between these two variables as well as there is no ordering dependency between x and r therefore, the constraints can be solved at the same time.

The solving order:

1. y
2. x and r
3. z

The constraints used for solving x, y, z and r are:

1. y is solved using the constraint y inside $\{2,3\}$.
2. x and r are solved using the constraints $x \% y == 0, x \neq y+r$ and $r == 5$.
3. z is solved using the constraints $z == x * y$.

A warning message will be issued for the following usage of `$void()`:

- The entire constraint expression being the parameter to `$void()` (for example, constraint `b1 { $void(x < y); }`)
 - The runtime will ignore the `$void()` specification.
- Expressions with only constants and state variables, including loop variables in the case of array constraints, being parameters to `$void()` (for example, constraint `b1 { x == $void(5); }`)
 - the runtime will ignore the `$void()` specification.

- A `void()` function being applied to a sub-expression of the parameter of another `void()` function (for example, `constraint b1{x==$void(y + $void(z));}`
 - the runtime will ignore the nested `$void()` specifications (that is, same as `x==$void(y+z)`).

Whenever the compile is not able to detect such conditions, no warning is issued at runtime.

solve-before hard. The solve-before construct allows for an optional **hard** modifier.

Example 24-3

```
constraint b1{
    y inside (2,3);
    x%y == 0;
    solve y before x hard;
}
```

The use of hard solve-before imposes an ordering mechanism between the variables. You can use ordering to assign a priority order to each variable. A higher priority indicates that the variable should be solved earlier.

Array Variable: If the argument to solve before hard is an array, the array is expanded into its individual elements before ordering is imposed which is similar to solve-before operation. See [page 97](#) for examples on how this happens.

The set of variables incident on a constraint is split into two sets – those at the lowest priority among the incident variables and those not at the lowest priority. Then the following semantic rules apply:

1. The constraint is not considered when any but the lowest priority variables incident on it are solved (that is, the higher priority variables are solved before the constraint is solved).

2. The constraint is solved when the lowest priority variables incident on it are solved.
3. When the constraint is solved, all lowest priority variables incident on it get solved.
4. hard solve-before constraints in OFF constraint blocks are ignored, and do not affect partitioning in any way.

Example 24-4

```

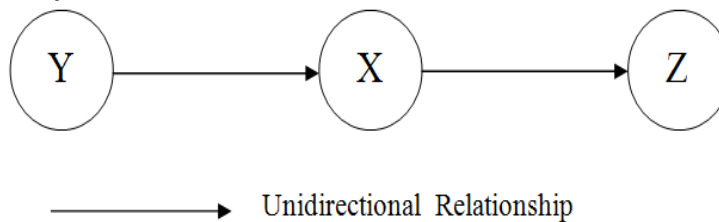
constraint b1 {
    y inside {2,3};
    x % y == 0;
    x != y;
    z == x * y;
    solve y before x hard;
    solve x before z hard;
}

```

Table 24-3

Constraint	Early	Late
<code>y in{2,3};</code>		y
<code>x%y == 0;</code>	y	x
<code>x !=y;</code>	y	x
<code>z==x*y;</code>	x, y	z

Figure 24-2 Dependencies between variables



The constraint(s) used for solving x, y and z:

1. y is solved using the constraint: `y inside (2,3);`

2. x is solved using the constraints: $x \% y == 0$; $x != y$;
3. z is solved using the constraint: $z == x * y$;

The sequence in which the constraints are solved is exactly as listed in the previous section.

solve-hard before and \$void(). `$void()` and solve-before hard have the same behavior regarding partitioning the solver-space (that is, every randomized variable inside the `$void()` or solve-before hard is solved first and set to static values prior to randomizing any other variables in the constraint).

The difference is that you can use the void construct as part of the expression, whereas solve-before hard is on a separate line, by itself.

Effect of rand_mode on Unidirectional Constraints. Using `rand_mode` to turn a variable OFF has the same effect as declaring the variable as non-random. Thus, ordering directives on it are dropped. This results in the dropping some transitive ordering relationships as well. For example in:

```
constraint b1 {
    y inside {2,3};
    x % $void(y) == 0;
    x != y; z == $void(x);
}
```

If x were to be turned OFF, then there would be no ordering relationship left between y and z either.

Extensions to Semantics. The introduction of unidirectional constructs has resulted in extensions to the semantics of numerous pre-existing constraint constructs.

Semantics of solve-before construct without the hard modifier

The solve-before constraints that are not modified by the “hard” directive will be used together in a partition. Thus, the non-hard solve-before constraints will not affect the partitioning, but will affect the order of variable assignment within a partition, very much like they affect the order of variable assigning on the complete problem prior to this release.

This is illustrated by the following example:

```
constraint b1 {
    y inside {2,3};
    x % y == 0;
    x != y;
    z == void(x * z);
    solve y, z before x;
}
```

The sequence in which the constraints are solved is:

1. x , y are solved using the constraints y inside $\{2,3\}$; $x \% y == 0$; solve y before x ;
2. z is solved using the constraint $z == x * y$; Note that since x is already solved at this point, the constraint solve z before x is irrelevant and hence dropped.

Semantics for Array Constraints (Solving Array Sizes)

Either of the two constructs defined above can be used to determine the partition to be solved when solving for the size of an array. The following example is used to illustrate the semantics:

```

class B;
  rand integer x[$]; rand bit[1:0] k;
  constraint b1 {
    x.size() == k; k != 2;
    foreach(x, i) {
      $void(k-1) == i -> x[i] == 0;
    }
  }
endclass

```

The constraints `x.size() = k; k != 2;` are solved for, since the use of the `$void()` function call renders `k` unconstrained in the constraint inside the `foreach` loop.

The use of `$void()` in this example specifies that `k` is solved for before any of the members of the `x` array.

The following example illustrates the other important semantic:

Example 24-5

```

class B;
  rand integer x[$];
  rand bit[1:0] k;

  constraint b1 {
    x.size() == $void(k);
    k != 2;
    foreach(x, i) {
      k-1 == i -> x[i] == 0;
    }
  }
endclass

```

In this case, `k` has to be solved before `x.size()`, and `x.size()` needs to be solved before expanding the `foreach` loop. Hence, there is an implicit ordering between `x.size()` and `x[i]`, requiring that `x.size()` gets solved first. As a result, the sequence in which the constraints will be solved:

1. `k` is solved for using the constraints `k != 2`.
2. `x.size()` is solved for using the constraints `x.size() == k`.

3. The loop is expanded and all the members of the `x` array are solved for.

The semantics of array constraints in the presence of `$void` calls are as follows:

1. Implicit ordering relations are introduced between array sizes and members of the array, *only* those with variable expressions in the index.
- Array sizes are solved before the constraints inside loops are solved or constraints involving array aggregate methods calls are solved.

Semantics of Default Constraints

With the introduction of partitions, it is possible that some random variables over which constraints are defined are solved before the default constraint itself. In such a case, the solver will consider previously solved random variables like state variables. When the default constraint is solved, it is disabled only if any of the currently being solved random variables incident on it are overridden. This is consistent with current semantics of default constraints, when considered in the context of the partition being solved.

Consider this example:

Example 24-6

```
default constraint b1 {
    x <= y;
}
constraint b1 {
    z == $void(y);
    z => x == 0;
    y inside {0, 1};
}
```

The sequence in which the constraints are solved is:

1. y is solved using the constraint `y inside {0,1}`.
2. z , x are solved using the default constraint `x<=y` and the non-default constraints `z==y; z=>x ==0;`. the default constraint applies if and only if z is 0 (that is, if and only if y was chosen as 0 in the previous partition.).

Semantics of `randc`

There is an existing semantic of the solver, wherein `randc` variables are solved in partitions that contain a single `randc` variable, after which they behave like state variables when solving for non-`randc` `rand` variables.

Example 24-7

```

randc bit[3:0]x;
rand bit[3:0] y,z;

constraint b1{
    y inside {0,1};
    x<12;
    if(y){
        x<5;
    } else{
        x<10;
    }
    z>x;
}

```

The sequence in which the above constraints are solved is:

1. x is solved for using the constraint `x<12`.
2. y and z are solved for using the remaining constraints. Note that this could result in a failure, depending on the solution picked in the previous step. For example, if x were picked as 11 in the previous step, then there would be no solution for y in this step.

Thus, all `randc` variables are solved before all `rand` variables.

However, when unidirectional constraints are used in the presence of `randc` variables, it is possible that `rand` variables could be solved before `randc` variables.

Example 24-8

```
randc bit[3:0]x;
rand bit[3:0] y,z;

constraint b1{
    y inside {0,1};
    x<12;
    if($void(y)){
        x<5;
    } else{
        x<10;
    }
    z>x;
}
```

The sequence in which the constraints are solved is:

1. `y` is solved using the constraint `y inside {0,1}`.
2. `x` is solved using the constraints `if(y) x<5 else x<10;`
`x<12`.
3. `z` is solved using the constraint `z>x`.

Static Constraint Blocks

The keyword `static` preceding the keyword `constraint`, makes a constraint block static, and calls to the `constraint_mode()` method occur to all instances of the constraint in all instances of the constraint's class.

```
static constraint c1 { data1 < data2;}
```

Randomize Methods

randomize()

Variables in an object are randomized using the `randomize()` class method. Every class has a built-in `randomize()` method.

```
randomize()
```

Generates random values for active random variables, subject to active constraints, in a specified instance of a class. This method returns a 1 if VCS generates these random values, otherwise it returns 0. Examples of the use of this method appear frequently in previous code examples.

pre_randomize() and post_randomize()

Every class contains built-in `pre_randomize()` and `post_randomize()` tasks, that are automatically called by `randomize()` before and after it computes new random values.

You may override the `pre_randomize()` method in any class to perform initialization and set pre-conditions before the object is randomized. Also, you may override the `post_randomize()` method in any class to perform cleanup, print diagnostics, and check post-conditions after the object is randomized.

The following example involves a case where the `pre_randomize()` method is used to instantiate and initialize the `rand` dynamic array, `payload[]`, before `randomize()` is called. The `post_randomize()` method is used to display the size of the `payload[]` after `randomize()` is called.

```
program test;
    class size;
        rand bit [7:0] s; //random variable s
```

```

        constraint con_size {
            s > 0;
            s < 50;
        }
    endclass

class Pkt;
    integer header[7];
    rand bit [7:0] payload[];
    size sz;
    function void pre_randomize();
    /* Randomize the sz.s variable and instantiate
    the dynamic array. */
        integer res;
        sz = new();
        res = sz.randomize(); /* calling randomize()
        on object sz of class size. Randomizes rand
        variable s (constrained to >0 and <50 */

        if(res==0)
            $display("Randomization Failed");
        else
            begin
                if((sz.s>0) && (sz.s<50))/* check for
                size between 0 and 50*/
                    payload = new[sz.s];/* the new[] operator
                    allocates storage and initializes the
                    rand variable s in sz*/
                else
                    $display("Failed to generate proper
                    size");
            end //end of outer else block
    endfunction

    function void post_randomize();
    /* display size of payload dynamic array using
    the size() built-in method*/
        $display("payload.size = %d", payload.size);
    endfunction
endclass

Pkt pkt;

```

```

integer success,i;

initial begin
    pkt = new(); //instantiate pkt

    for (int j = 0; j < 5; j++)
    begin
        success = pkt.randomize(); /*calling randomize
            on object of class packet*/
        if(success==0)
            $display("Randomization Failed");
        else
            $display("Randomization Succeeded");
    end // end of for loop
end // end of initial block
endprogram

```

The output of the program is:

```

payload.size =          12
Randomization Succeeded
payload.size =          17
Randomization Succeeded
payload.size =           8
Randomization Succeeded
payload.size =          40
Randomization Succeeded
payload.size =          10
Randomization Succeeded

```

Controlling Constraints

The predefined `constraint_mode()` method can be used either as a task or a function. The `constraint_mode()` task controls whether a constraint is active or inactive. The predefined `constraint_mode()` function reports the current ON/OFF value for the specified variable. All constraints are initially active.

Syntax:

```
task object[.constraint_identifier]::constraint_mode  
    (bit ON | OFF);
```

or

```
function int  
    object.constraint_identifier::constraint_mode();
```

In the following example, there are two constraint blocks defined in a class, `bus`. The `constraint_mode()` method, when used as a task, turns on and off the `word_align` and `addr_range` constraints. `constraint_mode()` is also being used as a function to report the ON/OFF value of the two constraints.

```
`define N 5  
program test;  
class bus;  
    rand bit [15:0] addr;  
    constraint word_align {addr[0] == 1'b0;}  
    constraint addr_range{addr >= 0 && addr <= 15;}  
endclass  
  
task generateRandomAddresses(integer how_many);  
    integer i;  
    for(i = 1; i <= how_many; i++) begin  
        bb.randomize();  
        $display("bb.addr = %d",bb.addr);  
    end  
endtask  
  
bus bb = new;  
initial begin  
  
    // By default all constraints are ON  
    if (bb.word_align.constraint_mode() &&  
        bb.addr_range.constraint_mode())  
        begin  
            $display("====both constraints ON====");  
        end
```

```

        else
            $display("Error with constraint_mode");

generateRandomAddresses(`N);

// turn OFF "word_align" constraint in "bb"
bb.word_align.constraint_mode(0);
$display("====one constraint ON =====");
generateRandomAddresses(`N);

// turn OFF all constraints in "bb"
bb.constraint_mode(0);
$display("====both constraints OFF =====");
generateRandomAddresses(`N);

// turn ON "addr_range" constraint in "bb"
bb.addr_range.constraint_mode(1);
    $display("====one constraint ON =====");
    generateRandomAddresses(`N);

    // turn ON all constraint in "bb"
    bb.constraint_mode(1);
    $display("====both constraints ON =====");
    generateRandomAddresses(`N);
end
endprogram
Output of the above program:
====both constraints ON =====
bb.addr =    14
bb.addr =     2
bb.addr =     4
bb.addr =     8
bb.addr =     2
====one constraint ON =====
bb.addr =     2
bb.addr =     9
bb.addr =     9
bb.addr =     9
bb.addr =     3
====both constraints OFF =====
bb.addr = 44051

```

```

bb.addr = 36593
bb.addr = 19491
bb.addr = 6853
bb.addr = 48017
=====one constraint ON =====
bb.addr = 11
bb.addr = 8
bb.addr = 15
bb.addr = 7
bb.addr = 4
=====both constraints ON =====
bb.addr = 2
bb.addr = 10
bb.addr = 12
bb.addr = 10
bb.addr = 8

```

Disabling Random Variables

SystemVerilog provides the predefined `rand_mode()` method to control whether a random variable is active or inactive. Initially, all random variables are active.

The `rand_mode()` method can be used either as a task or a function.

The `rand_mode()` task specifies whether or not the random variable is ON or OFF.

```

task object [.randvar_identifier]::rand_mode(bit ON |
      OFF);

```

where *ON* is “1” and *OFF* is “0.”

As a function, `rand_mode()` reports the current value (ON or OFF) of the specified variable.

Syntax:

```
function int object.randvar_identifier::rand_mode();
```

`rand_mode()` returns -1 if the specified variable does not exist within the class hierarchy or it exists but is not declared as `rand` or `randc`.

In the following example, there are two random variables defined in a class, `bus`. The `rand_mode()` method, when used as a task, turns on and off the random variables, `addr` and `data`. `rand_mode()` is also being used as a function to report on the value (ON or OFF) of the two random variables.

```
`define N 5
program test;
  class bus;
    rand bit [15:0] addr;
    rand bit [31:0] data;
    constraint CC { data > 0; addr > 0; addr < 255; data
      < 512;}
  endclass

  task generateRandomAddresses(integer how_many);
    integer i;
    for(i = 1; i <= how_many; i++) begin
      bb.randomize();
      $display("bb.addr = %d,, bb.data =
        %d",bb.addr,bb.data);
    end
  endtask

  bus bb = new;

  initial begin
    // By default all random variables are ON
    if (bb.addr.rand_mode() && bb.data.rand_mode())
      begin
        $display("====both random variables ON
          =====");
      end
    else
      $display("Error with rand_mode");
  end
end
```



```

generateRandomAddresses(`N);

// turn OFF "data" random variable in "bb"
bb.data.rand_mode(0);
$display("====one random variable ON =====");
generateRandomAddresses(`N);

// turn OFF all random variables in "bb"
bb.rand_mode(0);
$display("====both random variables OFF=====");
generateRandomAddresses(`N);

// turn ON "data" constraint in "bb"
bb.data.rand_mode(1);
$display("====one random variable ON=====");
generateRandomAddresses(`N);

// turn ON all random variable in "bb"
bb.rand_mode(1);
$display("====both random variables ON =====");
generateRandomAddresses(`N);
end
endprogram

```

Output of the above program:

```

====both random variables ON =====
bb.addr = 88,, bb.data = 12
bb.addr = 49,, bb.data = 381
bb.addr = 185,, bb.data = 5
bb.addr = 212,, bb.data = 486
bb.addr = 49,, bb.data = 219
====one random variable ON =====
bb.addr = 3,, bb.data = 219
bb.addr = 130,, bb.data = 219
bb.addr = 26,, bb.data = 219
bb.addr = 90,, bb.data = 219
bb.addr = 121,, bb.data = 219
====both random variables OFF=====
bb.addr = 121,, bb.data = 219

```

```

bb.addr = 121,, bb.data = 219
bb.addr = 121,, bb.data = 219
bb.addr = 121,, bb.data = 219
bb.addr = 121,, bb.data = 219
=====one random variable ON=====
bb.addr = 121,, bb.data = 456
bb.addr = 121,, bb.data = 511
bb.addr = 121,, bb.data = 67
bb.addr = 121,, bb.data = 316
bb.addr = 121,, bb.data = 405
=====both random variables ON =====
bb.addr = 18,, bb.data = 491
bb.addr = 231,, bb.data = 113
bb.addr = 118,, bb.data = 230
bb.addr = 46,, bb.data = 96
bb.addr = 155,, bb.data = 298

```

In-line Constraints

You can use the `randomize()` method and the `with` construct to declare in-line constraints outside of the class for the random variables, at the point where you call the `randomize()` method.

```

program prog;

class Bus;
    rand bit [2:0] bitrand1;
endclass

Bus bus = new;

task inline (Bus bus);
    int int1;

```

```

repeat (10)
  begin
    int1 = bus.randomize() with {bitrand1[1:0] ==
      2'b00};
    if (int1 ==1) $display("bitrand1 = %0b",
      bus.bitrand1);
  end
endtask

initial
inline(bus);

endprogram

```

The `$display` system task displays the following

```

bitrand1 = 100
bitrand1 = 100
bitrand1 = 100
bitrand1 = 0
bitrand1 = 100
bitrand1 = 100
bitrand1 = 0
bitrand1 = 100
bitrand1 = 100
bitrand1 = 0

```

In-line Constraint Checker

The `randomize()` method can act as a checker when a special “null” argument is specified. In this case, `randomize(null)` assigns no random values. It only returns a status. The status “1” is returned if all constraints are satisfied. A “0” is returned otherwise. The in-line random variable control mechanism can also be used to force the `randomize()` method to behave as a checker.

The following example illustrates the usage of in-line constraint checker.

```
program test;

class myclass;
    rand int a, b; // random variable a,b

    constraint cc { a + b == 4;}

    function new(); // constructor initializing values of a
                    // and b
        a = 2;
        b = 2;
    endfunction
endclass

myclass mc = new;
int i;
initial begin

    $display("Calling Randomize method with null args :-");
    i = mc.randomize(null); // values of a and b should be
                            // preserved
    $display("a = %0d, b = %0d, return from method = %0d",
            mc.a, mc.b, i); // should display a and b as 2
end
endprogram
```

The output of this program is:

```
Calling Randomize method with null args :-
a = 2, b = 2, return from method = 1
```

Random Number Generation

SystemVerilog includes a number of system functions to work with random numbers. Some of these are:

\$urandom()

The system function `$urandom()` generates pseudorandom numbers. It returns an unsigned 32-bit number every time this function is called. The syntax is:

```
$urandom( seed )
```

The `seed` is an optional argument that determines the sequence of the random number that are generated. It could also be an expression. The same sequence is always generated for the same seed.

The following program explains the usage of `$urandom()`.

```
program test();

bit [7:0] a,b,c,d;

initial begin
    $display("Generation of random number with seed 3 :");
    a = $urandom(3);
    b = $urandom();
    c = $urandom();
    d = $urandom();
    $display("a = %0d,b= %0d, c = %0d, d = %0d",a,b,c,d);

    $display("Generation of random number with seed 4 :");
    a = $urandom(4);
    b = $urandom();
    c = $urandom();
    d = $urandom();
    $display("a = %0d,b= %0d, c = %0d, d = %0d",a,b,c,d);
end
```

```

        $display("Generation of random number with seed 3 :");
        a = $urandom(3);
        b = $urandom();
        c = $urandom();
        d = $urandom();
        $display("a = %0d,b= %0d, c = %0d, d = %0d",a,b,c,d);
    end
endprogram

```

The output of this program is:

```

Generation of random number with seed 3 :
a = 229,b= 175, c = 7, d = 99
Generation of random number with seed 4 :
a = 228,b= 15, c = 254, d = 230
Generation of random number with seed 3 :
a = 229,b= 175, c = 7, d = 99

```

\$urandom_range()

The system function `$urandom_range()` generates random numbers within a certain range. The syntax is:

```
$urandom_range( unsigned int maxval, unsigned int minval)
```

The function returns an unsigned integer between `maxval` and `minval` (inclusive of them). If one of the arguments is omitted, then the function will return an unsigned integer between the argument specified and zero.

The following program explains the usage of `$urandom_range()`.

```

program test();

    logic [3:0] a,a1;
    bit [2:0] b;

    initial begin

```

```

    a = 10; a1 = 3;
$display("Generating random numbers with urandom_range
with expressions in range");
//generating value of b between 7 and 3
b = $urandom_range((a-a1), a1); //expressions as
args to urandom_range
$display("value of b generated with range %0d and
%0d is %0d",a-a1,a1,b);

//generating value of b between 3 and 7
b = $urandom_range(a1, (a-a1));
$display("value of b generated with range %0d and
%0d is %0d",a1,a-a1,b);

$srandom(2);    // changing the seed for thread

//generating value of b between 0 and 7 (omitting
one of the range value)
b = $urandom_range(a-a1);
$display("value of b generated with range %0d and
%0d is %0d",0,a-a1,b);
end
endprogram

```

The output of this program is:

```

Generating random numbers with urandom_range with
expressions in range

```

```

value of b generated with range 7 and 3 is 6
value of b generated with range 3 and 7 is 7
value of b generated with range 0 and 7 is 0

```

\$srandom()

You can use the system function \$srandom() to manually set the seed for random number generation. The syntax is:

```

$srandom(seed)

```

Providing the same seed ensures that the same set of random values are generated by the random number generator.

The following program demonstrates that the same random numbers are generated for the same seed.

```
program test;

logic [3:0] a, b;

initial begin
    $display("Setting the seed as 2 through srandom");
    $srandom(2);
    repeat (2)
        begin
            a = $urandom();
            b = $urandom();
            $display("a = %0d, b = %0d", a, b);
        end // end of repeat block
    $display("Changing the seed as 1 through srandom");
    $srandom(1);
    repeat (2)
        begin
            a = $urandom();
            b = $urandom();
            $display("a = %0d, b = %0d", a, b);
        end // end of repeat block
    $display("Setting the seed once again to 2 through
        srandom");
    $srandom(2);
    repeat (2)
        begin
            a = $urandom();
            b = $urandom();
            $display("a = %0d, b = %0d", a, b);
        end // end of repeat block
end // end of initial block
endprogram
```


The output of this program is:

```
Setting the seed as 2 through srandom
a = 8, b = 3
a = 14, b = 3
Changing the seed as 1 through srandom
a = 12, b = 13
a = 15, b = 9
Setting the seed once again to 2 through srandom
a = 8, b = 3
a = 14, b = 3
```

Seeding for Randomization

The `srandom()` method initializes the current Random Number Generator (RNG) of objects or threads using the value of the seed.

The following example involves using `srandom()` to initialize the RNG for an object using a real variable as the seed.

```
program test;

class A;
    rand logic [7:0] x;
endclass

real r = 1;

A a;
int d1,d2,d3,d4;
initial begin
    a = new;
    a.srandom(r); //the r is the seed for RNG of a
    d1 = a.randomize();
    if(d1 == 1) //if randomize() is successful
    d2 = a.x; //assign value of the variable x in a to d2
    a.srandom(r+1);
    d1 = a.randomize();
    if(d1 == 1)
```

```

    d3 = a.x;
    a.srandom(r);
    d1 = a.randomize();
    if(d1 == 1)
        d4 = a.x;
        if((d2 == d4) && (d2 != d3))
            $display("test passed");
    else
        $display("test failed");
end
endprogram

```

Output of the above program is:

```
test passed
```

randcase Statements

The randcase construct specifies a block of statements with corresponding weights attached to them. Weights can be any expression. When a randcase statement occurs, one of the statement with weights is executed at random. If the weight is an expression, then it is evaluated each time the randcase statement is executed. If the weight is zero, then that branch is not executed. The randcase statements can be nested.

```

program test;

    int a= 2;
    logic [3:0] b;

    //variables to keep count of how many time a particular case
    //is executed in randcase
    int count_2,count_c;

    function f1();

```

```

    randcase

    a+3   : count_2++; // simple add expression as weight

    10    : count_c++; // constant used as weight

    endcase
endfunction

initial begin
    repeat (15)
        f1();
        $display("Number of times a+3 called is %0d", count_2);
        $display("Number of times constant called is %0d",
            count_c);
    end
endprogram

```

The output of this program is:

```

Number of times a+3 called is 5
Number of times constant called is 10

```

This example defines a randcase block of two statements with different weights (one as an expression and another as an integer). The probability that any single statement will be selected for execution is determined by the formula $\text{weight}/\text{total_weight}$. Therefore, in this example, the probability of the first statement being executed is 0.33, and the second statement is 0.66.

Random Sequence Generation

Note:

Random sequence generation is an LCA feature.

SystemVerilog's sequence generation allows you to specify the syntax of valid sequences using BNF-like notation. Random sequences are ideal for generating streams of instructions for which it is easier to specify the syntax of valid streams than the constraints on specific values.

This section includes the following:

- RSG Overview
- Production Declaration
- Production Controls

RSG Overview

The syntax for programming languages is often expressed in Backus Naur Form (BNF) or some derivative thereof. Parser generators use this BNF to define the language to be parsed. However, it is possible to reverse the process. Instead of using the BNF to check that existing code fits the correct syntax, the BNF can be used to assemble code fragments into syntactically correct code. The result is the generation of pseudo-random sequences of text, ranging from sequences of characters to syntactically and semantically correct assembly language programs.

SystemVerilog's implementation of a stream generator, the RSG, is defined by a set of rules and productions encapsulated in a `randsequence` block.

The general syntax to define a RSG code block is:

```
randsequence ([production_identifier])  
    production {production}  
endsequence
```

Note:Nesting of randsequence blocks is not supported

When the randsequence block is executed, random production definitions are selected and streamed together to generate a random stream. How these definitions are generated is determined by the base elements included in the block.

Any RSG code block is comprised of production definitions. Native Testbench also provides weights, production controls, and production system functions to enhance production usage. Each of these RSG components is discussed in detail in subsequent sections.

Production Declaration

A language is defined in BNF by a set of production definitions. The syntax to define a production is:

```
[function_datatype] production_identifier  
  [(tf_port_list)] : rs_rule{ |rs_rule};
```

production_identifier

Is the reference name of the production definition.

tf_port_list

task/function port list.

rs_rule

The syntax for *rs_rule* is:

```
rs_production_list [ := weight_specification  
  [rs_code_block]
```

rs_production_list

The syntax for *rs_production_list* is:

```
rs_prod{rs_prod} | rand join [(expression)]  
    production_item production_item {production_item}
```

weight_specification

The syntax for *weight_specification* is:

```
integral_number | ps_identifier | (expression)
```

(see also “Weights for Randomization” on page 24-122)

rs_code_block

The syntax for *rs_code_block* is:

```
{{data_declaration}{statement_or_null}}
```

The following table provides the syntax for the non-terminals within *rs_production_list*, *weight_specification*, and *rs_code_block*, and the non-terminals therein.

non-terminal	Syntax
<i>rs_prod</i>	<i>production_item</i> <i>rs_code_block</i> <i>rs_if_else</i> <i>rs_repeat</i> <i>rs_case</i>
<i>rs_code_block</i>	{{ <i>data_declaration</i> }} { <i>statement_or_null</i> }
<i>rs_if_else</i>	if(<i>expression</i>) <i>product_item</i> [else <i>production_item</i>] see also “if-else Statements” on page 24-123
<i>rs_repeat</i>	repeat(<i>expression</i>) <i>production_item</i> see also “repeat Loops” on page 24-126
<i>rs_case</i>	case(<i>expression</i>) <i>rs_case_item</i> { <i>rs_case_item</i> } endcase see also “case Statements” on page 24-125
<i>re_case_item</i>	<i>expression</i> { <i>expression</i> } : <i>production_item</i> default [:] <i>production_item</i> ; see also “case Statements” on page 24-125
<i>production_item</i>	<i>production_identifier</i> [(<i>list_of_arguments</i>)]

A randsequence block is made up of one or more *productions*, yielding a “production list.”

Productions are made up of terminals and non-terminals, as seen in the above syntax description. A terminal is an indivisible code element. It needs no further definition beyond the code block associated with it. A non-terminal is an intermediate variable defined in terms of other terminals and non-terminals. If a production item is defined using non-terminals, those non-terminals must then be defined in terms of other non-terminals and terminals using the production definition construct. Ultimately, every non-terminal has to be broken down into its base terminal elements.

Multiple production items specified in a production list can be separated by white space or by the or operator (|). Production items separated by white space indicate that the items are streamed together in sequence. Production items separated by the | operator force a random choice, which is made every time the production is called.

The following is an example of a program including a randsequence block.

```
program p;
initial begin
    randsequence()
    main : repeat (10) TOP;
    TOP: RJ {$display("");};
    RJ: rand join (1.0) S1 S2 S3;
    S1 : A B;
    S2 : C D;
    S3 : E F G;
    A : {$write ("A");};
    B : {$write ("B");};
    C : {$write ("C");};
    D : {$write ("D");};
    E : {$write ("E");};
end
```

```
F : {$write ("F");};  
G : {$write ("G");};  
endsequence  
end  
endprogram
```

Production Controls

SystemVerilog provides several mechanisms that can be used to control productions: weights for randomization, if-else statements, case statements, and repeat loops.

This section includes:

- Weights for Randomization
- if-else Statements
- case Statements
- repeat Loops
- break Statement

Weights for Randomization

Weights can be assigned to production items to change the probability that they are selected when the randsequence block is called.

Syntax

```
rs_production_list[:= weight_specification [rs_code_block] ]
```


weight_specification

The syntax for `weight_specification` is:

```
integral_number | ps_identifier | (expression)
```

The expression can be any valid SystemVerilog expression that returns a non-negative integer. Function calls can be made within the expression, but the expression must return a numeric value, or else a simulation error is generated.

Assigning weights to a production item affects the probability that it is selected when the `randsequence` block is called. Weight should only be assigned when a selection is forced with the `|` operator. The weight for each production item is evaluated when its production definition is executed. This allows you to change the weights dynamically throughout a sequence of calls to the same production.

if-else Statements

A production can be conditionally referenced using an if-else statement.

The syntax to declare an if-else statement within a production definition is:

```
if(expression) product_item [else production_item]
```

expression

Can be any valid SystemVerilog expression that evaluates to a boolean value.

If the conditional evaluates to true, the first production item is selected. If it evaluates to false, the second production item is selected. The else statement can be omitted. If it is omitted, a false evaluation ignores the entire if statement. The following is an example of a production definition with an if-else statement.

```
assembly_block : if (nestingLevel > 10) seq_block else
                any_block;
```

This example defines the production `assembly_block`. If the variable `nestingLevel` is greater than 10, the production item `seq_block` is selected. If `nestingLevel` is less than or equal to 10, `any_block` is selected.

case Statements

A general selection mechanism is provided by the case statement. The syntax to declare a case statement within a production definition is:

```
case(expression)
    rs_case_item {rs_case_item}
endcase
```

expression

Is evaluated. The value of the *expression* is successively checked, in the order listed, against each *rs_case_item*. The production corresponding to the first matching case found is executed, and control is passed to the production definition whose name is in the case item with the matching case expression. If other matches exist, they are not executed. If no case item value matches the evaluated primary expression and there is no default case, nothing happens.

rs_case_item

Can be any valid SystemVerilog expression or comma-separated list of expressions. Expressions separated by commas allow multiple expressions to share the same statement block. The syntax is:

```
expression{,expression}: production_item | default [:]
    production_item;
```

A case statement must have at least one case item aside from the default case, which is optional. The default case must be the last item in a case statement. The following is an example of a production definition using a case statement:

repeat Loops

The repeat loop is used to loop over a production a specified number of times.

The syntax to declare a repeat loop within a production definition is:

```
repeat(express) production_item
```

expression

Can be any valid SystemVerilog expression that evaluates to a non-negative integer, including functions that return a numeric value.

The expression is evaluated when the production definition is executed. The value specifies how many times the corresponding production item is executed. The following is an example of a production definition using a repeat loop.

```
randsequence()  
    ...  
    seq_block : repeat (random() ) integer_instruction;  
    ...  
endsequence
```

This example defines the production `seq_block`, which repeats the production item `integer_instruction` a random number of times, depending on the value returned by the `random()` system function.

break Statement

The break statement is used to terminate a randsequence block.

Syntax

```
break;
```

A break statement can be executed from within an *rs_code_block* within a production definition. When a break statement is executed, the randsequence block terminates immediately and control is passed to the first line of code after the randsequence block. The following is an example of a production definition using a break statement.

return Statement

The **return** statement is used to interrupt the execution of the current production. After the current production is aborted, the execution continues on the next item in the production from which the call is made.

Syntax

```
return;
```

The **return** statement passes control to the next production item in the production from which the call is made without executing any code in between. The following is an example of a production definition using a **return** statement:

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) to 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 subclasse(s). 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 AOE.

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 24-157](#). 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 24-143](#).

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 24-157](#).

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 24-4 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 ae1\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 ae1\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 24-9

Target method:

```
task myTask();
    $display("Executing original code\n");
endtask
```

Advice:

```
after task myTask ();
    $display("Before in ae1\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 aoel\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 aoel\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 24-10

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 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(). 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 24-157](#).

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 24-134) 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 24-11

```
// 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 24-11, 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 24-11 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 24-12

```
// Beginnning 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 24-12 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 24-13

```

// 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 24-13 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 24-14

```
// Beginnning 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 24-14 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 24-15

```

// Beginnning 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 24-15 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 24-16

```

// 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 aoel (foo);
    around task myTask();
        proceed;
        printf("around in aoel\n");
    endtask
endextends
extends aoe2 (foo) dominates (aoel);
    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 24-16, the output is:

```

Executing original code
around in aoel
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 possible for AOE code to reference symbols that were added to the original class definition using AOE's.

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 24-17

```
// 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 24-17 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 24-18

```

// 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 24-18, 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 **virtualls** 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 24-19, the *print* method introduced by the *extends* directive hides the *print* method in the super class.

Example 24-19

```
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 24-20 illustrates how the two hide permissions can change at different aspect items within an aspect extension.

Example 24-20

```
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 24-21 is 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 24-21

```
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 24-22 will is not allowed, as the aspect *myaspect* defines *x* as one of the introductions when the symbol *x* is already defined in class *foo*.

Example 24-22 : 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 24-23**, the extends directive adds advices to the packet::send method.

Example 24-23 :

```

// 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 24-23 is executed, the output is:

```

Before sending packet
Sending data
After sending packet

```


In Example 24-24, extends directive myaspect adds advice to turn off constraint c1 before each call to the foo::pre_randomize method.

Example 24-24 :

```
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 24-23, extends directive myaspect adds advice to set a property named valid to 0 after each call to the foo::post_randomize method.

Example 24-25 :

```
class foo;
  integer valid;
  rand integer myfield;
  constraint c1 {
    myfield == 4;
  }
endclass

extends myaspect(foo);
  after task post_randomize();
    valid = 0;
  endtask
endextends
```

Example 24-25 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 24-26

```
// 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 24-26 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
```

Array manipulation methods

VCS-NTB provides the following types of built-in methods for analyzing and manipulating arrays.

- Array ordering methods
- Array locator methods
- Array reduction methods

Note: For the integral types the `with` construct is not required but for non integral type like class objects, mailbox or semaphores using the `with` construct enhances the array manipulation methods ability to deliver.

Array ordering methods

The array ordering methods are used to change the order and rearrange the elements of any arrays.

reverse()

The `reverse()` method puts all the elements in the reverse order. Specifying a `with` construct here will result in a compiler error.

Syntax

```
task array_name.reverse();
```

Example 24-27

```
int SQint[$] = {1, 2, 3, 4}; //SQint contains 1, 2, 3, 4
SQint.reverse(); // SQint contains 4, 3, 2, 1
```

sort()

The `sort()` method sorts the element in an ascending order using the expression in the `with` construct. The `with` expression is optional here.

Syntax

```
task array_name.sort() [with (expression)];
```

Example 24-28

```
queue1.sort();  
queue2.sort() with (item.i );
```

In the first example the `queue1` is sorted in ascending order of the value of the elements provided they are of an integral type. In the second example `queue2` is a queue of class objects and the class contains an element `i`, here the objects are sorted in ascending order depending on the value of the member variable `i`.

rsort()

The `rsort()` method sorts the elements in the array in a descending order using the expression in the `with` construct. The `with` expression is optional here.

Syntax

```
task array_name.rsort() [with (expression)];
```

Example 24-29

```
queue1.rsort();  
queue2.sort() with (item.i * item.r);
```

In the first example the elements in the `queue1` are sorted in descending order of their values provided they are of an integral type. In the second example `queue2` is a queue of class objects and the class contains an element `i` and `r`, now the objects are sorted in descending order depending on the product value of the member variable `i` and `r`.

Array locator methods

- Array locator methods
- Array index locator methods

The array locator methods are used to search all the elements in an array that satisfies a given expression. These methods return a queue containing all elements that satisfy the expression.

The array index locator methods are used to search an array for all the indices that satisfy a given expression and return an integer queue with the indices of all elements that satisfy the expression.

Note: For associative arrays the array index locator methods return a queue of their index type with the indices of all elements that satisfy the expression..

find()

The `find()` method finds all the elements satisfying the expression using the `with` construct. If the match fails or if the array is empty an empty queue is returned. The `with` expression is required and is not optional.

Syntax

```
function array_type[$] array_name.find() with (expression);
```

Example 24-30

```
queue2 = queue1.find() with (queue1.len() > 5);
```

In the example, `queue1` contains a set of strings, all the elements whose string length is greater than 5 are returned and they are assigned in the same order to the queue `queue2`.

find_index()

The `find_index()` method returns the indexes of all the elements satisfying the expression using the `with` construct. If the match fails or if the array is empty an empty queue is returned. The `with` expression is required and is not optional.

Syntax

```
function int_or_index_type[$] array_name.find_index()  
    with (expression);
```

Example 24-31

```
indices1 = queue1.find_index() with (item > 5);
```

In the example, all the indices of the elements that are greater than 5 are returned and they are assigned in the same order to the queue `indices1`.

find_first()

The `find_first()` method finds the first element satisfying the expression using the `with` construct. If the array is empty, a warning message is issued and an empty queue is returned. The `with` expression is required and is not optional.

Syntax

```
function array_type[$] array_name.first() with (expression);
```

Example 24-32

```
value = queue1.first() with (item > 5);  
object = queue1.first() with (item.x > 5);
```

In the first example, the first element which is greater than 5 is returned. In the second example, the first element with the member variable greater than 5 is returned.

find_first_index()

The `find_first_index()` method returns the first index satisfying the expression using `with` construct. If the array is empty or no match is found with the expression then an empty queue is returned. The `with` expression is required and is not optional.

Syntax

```
function int_or_index_type[$] array_name.find_first_index()  
                                         with (expression);
```

Example 24-33

```
index = queue1.first_index() with (item > 5);  
index = queue1.first_index() with (item.y > 5);
```

The first example returns the index of the first element which is greater than 5. In the second example, the index of the first element that contains the member variable `y` greater than 5 is returned.

find_last()

The `find_last()` method returns a queue with the last element satisfying the expression. If the array is empty, a warning message is issued and an empty queue is returned. The `with` expression is required and is not optional.

Syntax

```
function array_type[$] array_name.last() with (expression);
```

Example 24-34

```
value = array1.last() with (item == 5);  
object = array1.last() with (item.x == 5);
```

In the first example, the last element which is equal to 5 is returned. In the second example, the last element that has its member variable `x` equal to 5 is returned.

find_last_index()

The `find_last_index()` method returns a queue with the last index satisfying the expression. If the array is empty or no match is found with the expression then an empty queue is returned. The `with` expression is required and is not optional.

Syntax

```
function int_or_index_type[$] array_name.find_last_index()  
                                     with (expression);
```

Example 24-35

```
value = array1.find_last_index() with (item > 5);  
value = array1.find_last_index() with (item.x == 5);
```


In the first example, the index of the last element which is greater than 5 is returned. In the second example, the index of the last element has its member variable x equal to 5 is returned.

min()

The `min()` method returns a queue with the minimum value element for the integral types. For non-integral types the method returns a queue with the element whose `with` expression evaluates to a minimum. If the array is empty, a warning message is issued and a type dependent default value is returned. The `with` expression is optional here.

Syntax

```
function array_type[$] array_name.min() [with (expression)];
```

Example 24-36

```
value = array1.min(); // Array1 is of type integer or reg
object = array1.min() with (item.x >5);
object = array1.min() with (item.x * item.y);
```

The first example, returns a minimum value element of the array. The second example returns the element that has the minimum value of the member variable x . The third example returns the element that has the minimum value of the product of the member values of x and y .

max()

The `max()` method returns a queue with the maximum value element for the integral types. For non-integral types the method returns a queue with the element whose `with` expression evaluates to a maximum. If the array is empty a warning message is issued and a type dependent default value is returned. The `with` expression is optional here.

Syntax

```
function array_type[$] array_name.max() [with (expression)];
```

Example 24-37

```
value = array1.max();  
object = array1.max() with (item.x >5);  
object = array1.max() with (item.x * item.y);
```

The first example, returns a maximum value element of the array. The second example returns the element that has the maximum value of the member variable `x`. The third example returns the element that has the maximum value of the product of the member values of `x` and `y`.

unique()

The `unique()` method returns all elements with unique values or elements whose expression is unique. If the array is empty or if no match is found with the expression, then an empty queue is returned. The `with` expression is optional here.

Syntax

```
function array_type[$] array_name.unique() [with (expression)];
```

Example 24-38

```
array2 = array1.unique();
```

In the example only unique elements from the array are returned.

unique_index()

The `unique_index()` method returns the indexes of all elements with unique values or whose expression is unique. If the array is empty or if no match is found with the expression then an empty queue is returned. The `with` expression is optional here.

Syntax

```
function int_or_index_type[$] unique_index() [with (expression)];
```

Example 24-39

```
array2 = array1.unique_index();
```

In the example indices of only the unique elements are returned.

Array reduction methods

Array reduction methods are used to evaluate and reduce an array to a single value. The optional `with` clause can be used to specify the elements for reduction. These methods return a reduced single value as the array element type. If the `with` clause is specified, they return the type of the expression in the `with` clause.

sum()

The `sum()` method computes the sum of all the array elements for integral types. The `with` expression is optional for integral types. When the `with` expression is specified for non-integral types they returns the sum of the member variables specified using the `with` expression.

Syntax

```
function expression_or_array_type array_name.sum() [with (expression)]
```

Example 24-40

```
value = Qint.sum(); // Calculates sum of all elements
value = QPacket.sum() with (item.pkt_size); // Computes sum of
// member pkt_size for all elements of queue SQPacket.
```

In the first example the sum of all the elements of the queue `Qint` is calculated. The second example computes sum of member variable `pkt_size` for all elements of queue `QPacket` is calculated.

product()

The `product()` method computes the product of all the array elements for integral types. The `with` expression is optional for integral types. When the `with` expression is specified for non-integral types they returns the product of the member variables specified using the `with` expression.

Syntax

```
function expression_or_array_type array_name.product()
                                     [with (expression)];
```

Example 24-41

```
value = Qint.product(); // Calculates product of all elements
value = QPacket.product() with (item.pkt_size); // Computes
// product of member pkt_size for all elements of queue QPacket.
```

In the first example the product of all the elements of the queue `Qint` is calculated. The second example computes product of member variable `pkt_size` for all elements of queue `QPacket` is calculated.

and()

The `and()` method computes the bitwise AND (`&`) of all the array elements for integral types. The `with` expression is optional for integral types. When the `with` expression is specified for non-integral types they returns the bitwise AND (`&`) of the member variables specified using the `with` expression.

Syntax

```
function expression_or_array_type array_name.and() [with (expression)];
```

Example 24-42

```
value = Qint.and(); // Calculates bitwise AND of all elements
value = QPacket.and() with (item.pkt_size ); // Computes bitwise
// AND of member pkt_size for all elements of queue QPacket.
```

In the first example the bitwise AND of all the elements of the queue `Qint` is calculated. The second example computes bitwise AND of member variable `pkt_size` for all elements of queue `QPacket` is calculated.

or()

The `or()` method computes the bitwise OR (`|`) of all the array elements for integral types. The `with` expression is optional for integral types. When the `with` expression is specified for non-integral types they returns the bitwise OR (`|`) of the member variables specified using the `with` expression.

Syntax

```
function expression_or_array_type array_name.or() [with (expression)];
```

Example 24-43

```
value = Qint.or(); // Calculates sum of all elements
value = QPacket.or()with (item.pkt_size ); // Computes bitwise
// OR of member pkt_size for all elements of queue QPacket.
```

In the first example the bitwise OR of all the elements of the queue `Qint` is calculated. The second example computes bitwise OR of member variable `pkt_size` for all elements of queue `QPacket` is calculated.

xor()

The `xor()` method computes the logical XOR (^) of all the array elements for integral types. The `with` expression is optional for integral types. When the `with` expression is specified for non-integral types they returns the logical XOR (^) of the member variables specified using the `with` expression.

Syntax

```
function expression_or_array_type array_name.xor() [with (expression)];
```

Example 24-44

```
value = Qint.xor(); // Calculates XOR of all elements
value = QPacket.xor()with (item.pkt_size ); // Computes bitwise
// XOR of member pkt_size for all elements of queue QPacket.
```

In the first example the logical XOR of all the elements of the queue `Qint` is calculated. The second example computes logical XOR of member variable `pkt_size` for all elements of queue `QPacket` is calculated.

Interprocess Synchronization and Communication

Semaphores

SystemVerilog semaphores are not signal devices. They are buckets that contain keys, where competing resources, such as different initial blocks in a program, require keys from the bucket to continue processing.

```
program prog;

semaphore sem1 = new(2);

initial
begin:initial1
    #1 sem1.get(1);
    $display("initial1 takes 1 key at %0t", $time);
    #6 sem1.put(1);
    $display("initial1 returns 1 key at %0t", $time);
    #1 sem1.get(1);
    $display("initial1 takes 1 key at %0t", $time);

end

initial
begin:initial2
    #5 sem1.get(2);
    $display("          initial2 takes 2 keys at %0t", $time);
    #5 sem1.put(1);
    $display("          initial2 returns 1 key at %0t", $time);

end
endprogram
```

In this program there are two initial blocks, labeled by the label on their begin-end blocks, initial1 and initial2.

The program has a semaphore named `sem1` that starts with two keys, as specified with the `semaphore` keyword and `new()` method.

If it were not for `initial2`, `initial1` would do the following:

1. Take a key at simulation time 1 (using the `get` method).
2. Return a key at time 7 (using the `put` method).
3. Take a key again at time 8 (using the `get` method).

If it were not for `initial1`, `initial2` would do the following:

1. Take two keys at simulation time 5 (using the `get` method).
2. Return one key at time 10 (using the `put` method).

However both initial blocks contend for a limited number of keys that they need in order to finish executing, in taking keys that the other needs, they interrupt each other's processing. The `$display` system tasks display the following:

```
initial1 takes 1 key at 1
initial1 returns 1 key at 7
           initial2 takes 2 keys at 7
           initial2 returns 1 key at 12
initial1 takes 1 key at 12
```

The initial block `initial2` could be rewritten to use the `try_get` method to see if a certain number of keys are available, for example:

```
initial
begin:initial2
    #5 if(sem1.try_get(2))
        begin
            sem1.get(2);
            $display("initial2 takes 2 keys at %0t", $time);
        end
```



```

#5 sem1.put(1);
   $display("initial2 returns 1 key at %0t", $time);

end
endprogram

```

In the revised `initial2`, at simulation time 5, the `try_get` method checks to see if there are two keys in `sem1`. There aren't, because `initial1` took one. At time 10 the `put` method "returns" a key to `sem1`. Actually the `get` and `put` methods only decrement and increment a counter for the keys, there are no keys themselves, so `initial2` can increment the key count without having previously decrementing this count.

The `$display` system tasks display the following:

```

initial1 takes 1 key at 1
initial1 returns 1 key at 7
initial1 takes 1 key at 8
initial2 returns 1 key at 10

```

Semaphore Methods

Semaphores have the following built-in methods:

`new` (*number_of_keys*)

You use this method with the `semaphore` keyword. It specifies the initial number of keys in the semaphore.

`put` (*number_of_keys*)

Increments the number of keys in the semaphore.

```
get(number_of_keys)
```

Decrements the number of keys in the semaphore. If there aren't the specified number of keys in the semaphore, VCS halts simulation of the process (initial block, task, etc.) until there the `put` method in another process increments the number of keys to the sufficient number.

```
try_get (number_of_keys)
```

Decrements the number of keys in the semaphore. If there aren't the specified number of keys in the semaphore, this method returns a 0. If the semaphore has the specified number of keys, this method returns 1. After returning the value, VCS executes the next statement.

Mailboxes

Mailboxes are FIFO containers for messages that are expressions.

Note:The SystemVerilog 3.1a LRM specifies that you can specify a maximum number of messages that a mailbox can hold, but this feature isn't implemented yet.

```
program prog;

mailbox mbx = new ();
int i,j;
int k = 10;

initial
begin
repeat(3)
begin
#5 mbx.put(k);
i = mbx.num();
$display("No. of msgs in mbx = %0d at %0t",i,$time);
k = k + 1;
end
end
```

```

end
i = mbx.num();
repeat (3)
    begin
        #5 $display("No. of msgs in mbx = %0d j = %0d at
                    %0t", i, j, $time);
        mbx.get(j);
        i = mbx.num();
    end
end
endprogram

```

This program declares a mailbox named `mbx` with the `mailbox` keyword and the `new()` method.

The initial block does the following:

1. Executes a `repeat` loop three times which does the following:
 - a. Puts the value of `k` in the mailbox.
 - b. Assigns the number of messages in the mailbox to `i`.
 - c. Increments the value of `k`.
2. Execute another `repeat` loop three times that does the following:
 - a. Displays the number of messages in the mailbox, the value of `j`, and the simulation time.
 - b. Assigns the first expression in the mailbox to `j`.
 - c. Assigns the number of messages to `i`.

The `$display` system tasks display the following:

```

No. of msgs in mbx = 1 at 5
No. of msgs in mbx = 2 at 10
No. of msgs in mbx = 3 at 15
No. of msgs in mbx = 3 j = 0 at 20
No. of msgs in mbx = 2 j = 10 at 25
No. of msgs in mbx = 1 j = 11 at 30

```

Mailbox Methods

Mailboxes use the following methods:

`new()`

Along with the `mailbox` keyword, declares a new mailbox. You cannot yet specify the maximum number of messages with this method.

`num()`

Returns the number of messages in the mailbox.

`put(expression)`

Puts another message in the mailbox.

`get(variable)`

Assigns the value of the first message to the variable. VCS removes the first message so that the next message becomes the first method. If the mailbox is empty, VCS suspends simulation of the process (initial block, task, etc.) until a `put` method put a message in the mailbox.

`try_get(variable)`

Assigns the value of the first message to the variable. If the mailbox is empty, this method returns the 0 value. If the message is available, this method returns a non-zero value. After returning the value, VCS executes the next statement.

`peek(variable)`

Assigns the value of the first message to the variable without removing the message. If the mailbox is empty, VCS suspends simulation of the process (initial block, task, etc.) until a `put` method put a message in the mailbox.

```
try_peek(variable)
```

Assigns the value of the first message to the variable without removing the message. If the mailbox is empty, this method returns the 0 value. If the message is available, this method returns a non-zero value. After returning the value, VCS executes the next statement.

Note VCS does an assignment compatibility check instead of equivalent types check for the mailbox methods.

Events

SystemVerilog has a number of extensions to named events. These extensions are as follows:

Waiting for an Event

Persistent Trigger

Merging Events

Reclaiming Named Events

Event Comparison

Waiting for an Event

You can enter a hierarchical name for a named event in an event control.

```
`timescale 1ns/1ns  
program prog;
```

```

task t1;
event evt1;
#5 -> evt1;
endtask

initial
t1;

initial
@(t1.evt1) $display("t1.evt1 happened at %0t", $time);

endprogram

```

The `$display` system task displays the following:

```
t1.evt1 happened at 5
```

Persistent Trigger

The `triggered` property persists on a named event throughout the time step when it is triggered, preventing a race condition, for example, when a named event is triggered and is evaluated in an event control during the same time step.

```

program prog;

event evt1, evt2;

initial
-> evt1;

initial
begin
wait (evt1.triggered);
$display("evt1 triggered");
end

initial

```

```

fork
  -> evt2;
  begin
    wait (evt2.triggered);
    $display("evt2 occurred");
  end
join

endprogram

```

The `$display` system tasks display the following:

```

evt1 triggered
evt2 occurred

```

Merging Events

You can assign a SystemVerilog named event to another named event. When you do, they alias each other and when VCS executes a line calling for the triggering of one of these events, VCS triggers both named events.

```

program prog;

event evt1, evt2, evt3;

initial
begin
  evt2 = evt3; // this is an alias
  evt1 = evt3; // this is an alias
  #2 -> evt1;
end

initial
#1 @ (evt1) $display("evt1 triggered");

initial
#1 @ (evt2) $display("evt2 triggered");

```

```
initial
#1 @ (evt3) $display("evt3 triggered");

endprogram
```

The `$display` system tasks display the following:

```
evt1 triggered
evt2 triggered
evt3 triggered
```

IMPORTANT:

When you merge events, the merger takes effect only in subsequent event controls or `wait` statements.

In this example, the merging occurred at time 0, the event controls at time 1, and the triggering of the events at time 2.

Reclaiming Named Events

When you assign the `null` keyword to a named event, that named event no longer can synchronize anything. In an event control it might block forever or not at all. In a `wait` statement, it is as if the named event were undefined, and triggering the named event causes no simulation events.

```
program prog;
event evt1;

initial
begin
  evt1 = null;
  #5 -> evt1;
end

initial
```



```

#1 @(evt1) $display("evt1 triggered");

initial
begin
#5 wait (evt1.triggered);
$display("evt1 occurred");
end
endprogram

```

The `$display` system tasks do not display anything.

Event Comparison

You can use the equality and inequality operators to see if named events are aliased to each other or have been assigned the null value, for example:

```

program prog;

event evt1, evt2, evt3;

initial
begin
evt1 = evt2;
if (evt1 == evt2)
    $display("evt1 == evt2");
if (evt1 === evt2)
    $display("evt1 === evt2");
if (evt1 != evt3)
    $display("evt1 != evt3");
if (evt3 != null)
    $display("evt3 != null");
end
endprogram

```

The `$display` system tasks display the following:

```

evt1 == evt2

```

```
evt1 === evt2
evt1 != evt3
evt3 != null
```

In comparing named events, the case equality operator `===` works the same as the equality operator `==`, and the case inequality operator `!==` works the same as the inequality operator `!=`.

Clocking Blocks

A clocking block encapsulates a group of signals that are sampled or synchronized by a common clock. It defines the timing behavior of those signals with respect to the associated clock. Consequently, timing and synchronization details for these signals is separate from the structural, functional, and procedural elements of the testbench. This enables synchronous events, input sampling, and synchronous drives to be written without explicitly using clocks or specifying timing.

Clocking blocks can be declared inside a program block or inside an interface.

Clocking Block Declaration

The syntax for declaring a clocking block is:

```
clocking clocking_identifier @clocking_event;
    [default clocking_dir clocking_skew [clocking_dir
        clocking_skew];]
    {clocking_dir [clocking_skew][clocking_dir
        [clocking_skew]]signal_identifier [=
        hierarchical_identifier]
        {,signal_identifier [= hierarchical_identifier]};}
endclocking[: clocking_identifier]
```

clocking_identifier

Name of the clocking block being declared.

clocking_event

Event that acts as the clock for the clocking block (for example: posedge, negedge of a clocking signal):

```
@(posedge clk)
```

or

```
@(clk)
```

Note:

Program signals *cannot* be used inside a clocking event expression.

clocking_dir

Direction of the signal: input, output or inout. If you specify more than one *clocking_dir*, they must be in the order input...output:

```
input clocking_skew output clocking_skew
```

The inout signal cannot be used in the declaration of a default skew. Also, if the *clocking_dir* of a clocking block signal is inout, you cannot specify a *clocking_skew*. For example:

```
inout #1 d; //results in a syntax error
inout d;    //is fine
```

clocking_skew

Determines how long before the synchronized edge the signal is sampled, or how long after the synchronized edge the signal is driven. A *clocking_skew* can consist of an edge identifier and a delay control, just an edge identifier, or just the delay control. The edge identifiers are *posedge* and *negedge*. The edge can be specified *only* if the clocking event is a singular clock (that is, a simple edge of a single signal like `@(posedge clk)`, `@(clk)`, `@(negedge top.clk)`, etc.). The delay control is introduced by `"#"` followed by the delay value. The following are examples of legal *clocking_skews*:

```
input #0 i1;
output negedge #2 i2;
input #1 output #2;
```

Note:

Time literals (e.g., `#10ns` and `#2ns`) are not supported in this release.

The skew for an input signal is implicitly negative (that is, sampling occurs before the clock event). The skew for an output signal is implicitly positive (that is, the signal is driven after the clock event).

Note:

`#1step` is the default input skew unless otherwise specified. However, an explicit `#1step` skew is not yet supported.

signal_identifier

Identifies a signal in the scope enclosing the clocking block declaration, and declares the name of a signal in the clocking block. Unless a *hierarchical_expression* is used, both the signal and the *clocking_item* names shall be the same. For example:

```
input #1 i1;
```

where `i1` is the *signal_identifier*.

Note:

A clocking block signal can only be connected to a scalar, vector, packed array, integer or real variable. Program signals are not allowed in clocking block signal declarations.

hierarchical_identifier

Hierarchical path to the signal being assigned to the *signal_identifier*. For example:

```
input negedge #2 i = top.i2;
```

where `i2` is defined in a module `top`.

Note:

see page 182 of the System Verilog LRM 3.1a for a formal definition of the syntax for declaring the clocking block.

Note:

Slices and concatenations are not yet implemented

A single skew can be declared for the entire clocking block. For example:

```
default input #10;
```

You can override default skews when you declare a signal.

The following example includes a clocking block embedded in a program:

```
`timescale 1ns/1ns  
  
module top;
```

```

reg out3;
reg out1;
reg clk = 0;

p1 p(out3,clk,out1);

assign out1 = out3;

initial forever begin
    clk = 0;
    #10;
    clk = 1;
    #10;
end
endmodule

program p1(output reg out3,input logic clk,input reg in );

clocking cb @(posedge clk);
    output #3 out2 = out3; //CB output signal
    input #0 out1 = in;
endclocking

initial
    #200 $finish;

initial begin
    $display($time,,,cb.out1);
    cb.out2 <= 0;           //driving output at "0" time
        @(cb.out1);       //sampling input for change
    $display($time,,,cb.out1);
    #100;
    $display($time,,,cb.out1);

        cb.out2 <= 1;     //driving o/p at posedge of clk

            @(cb.out1);

                $display($time,,,cb.out1);

```

```
end  
  
endprogram
```

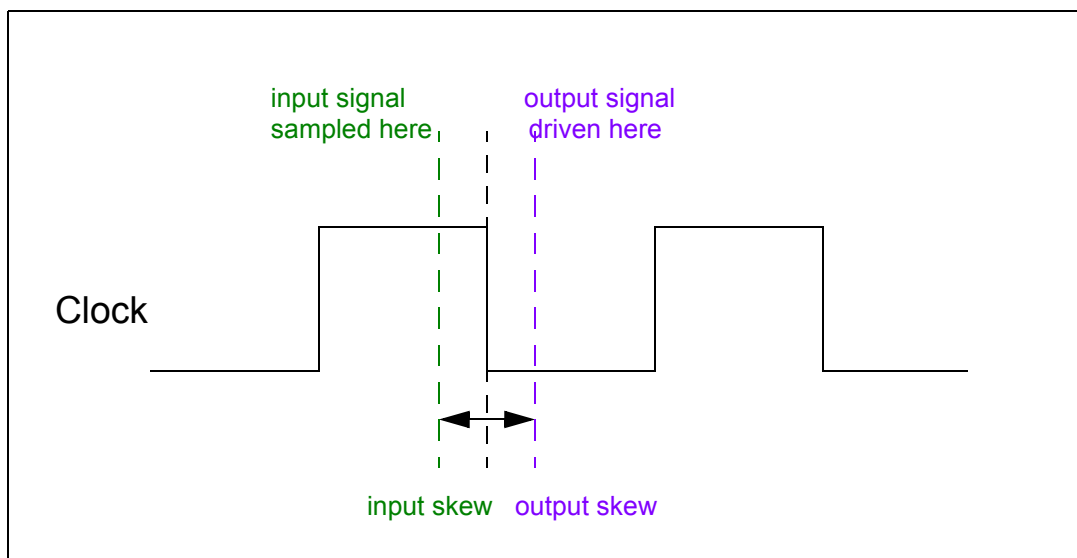
The output of this program is:

```
0  x  
30 0  
130 0  
150 1
```

Input and Output Skews

The skew for input and inout signals determines how long before *clocking_event* the signal is sampled. The skew for output and inout signals determines how long after the *clock_event* the signal is driven.

Figure 24-3 Driving and sampling on the negative edge of the clock



For more details see section 15.3 of the SystemVerilog LRM 3.1a.

Hierarchical Expressions

Every signal in a clocking block is associated with a program port or a cross module reference.

As described when defining *hierarchical_identifier*, the hierarchical path is assigned to the defined in the clocking block.

```
clocking cb2 @ (negedge clk);
    input #0 b = top.q;
endclocking
```

Below is an example of the *hierarchical_identifier* as a program port:

```
program p1(output reg out3,input logic clk,input reg in );
    clocking cb @(posedge clk);
        output #3 out2 = out3;//out3 and in = program ports
        input #0 out1 = in;
    endclocking
endprogram
```

Signals in Multiple Clocking Blocks

The signals (clocks, input, outputs, or inouts) associated with one clocking block can be associated with any other clocking blocks. For example:

```
program test( input bit clk_1, input bit clk_2,
             input reg [15:0] data );

default clocking data @(posedge clk_1);
    input data;
endclocking

clocking address @(posedge clk_2);
    input data;
```



```
endclocking
```

Clocking blocks that use the same clock can share the same synchronization event. For example:

```
program test( input bit clk_1, input reg [15:0] address,  
             input reg [15:0] data );
```

```
default clocking data @(posedge clk_1);  
    input data;  
endclocking
```

```
clocking address @(posedge clk_1);  
    input address;  
endclocking
```

Clocking Block Scope and Lifetime

Signals associated with a clocking block can be accessed by using a dot (.) operator. For example:

```
clocking CB_1 @(posedge clk_1);  
    input data;  
    input address;  
endclocking
```

```
CB_1.data;  
CB_1.address;
```

The scope of a clocking block is local to its enclosing module, interface, or program. Clocking blocks cannot be nested. They cannot be declared inside packages, functions, tasks, and outside all declarations in a compilation unit. The lifetime of a clocking block is static.

Clocking Block Events

The *clocking_identifier* can be used to refer to the *clocking_event* of a clocking block. For example:

```
clocking cb1 @(posedge clk);
    input #0 i1;
    input negedge #2 address;
endclocking
```

The clocking event of the `cb1` clocking block can be used to wait for that particular event:

```
@(cb1);
```

Therefore, `@(cb1)` is equivalent to `@(posedge clk)`.

Default Clocking Blocks

One clocking block can be specified as the default clocking block for all cycle delay operations within a given module, program, or interface.

Syntax:

```
default clocking clocking_identifier
```

or

```
clocking_declaration ::=
    [default] clocking [clocking_identifier] clocking_event;
endclocking
```

clocking_identifier

Name of a clocking block.

Note:

You can specify only one default clocking block in a program, module, or interface. VCS issues a compilation error if you specify more than one default clocking block.

For example:

```
program test( input bit clk, input reg [15:0] data );

default clocking bus @(posedge clk);
    input data;
endclocking

initial begin
    ## 5;
    if ( bus.data == 10 )
        ## 1;
    else
        ##2;
end

endprogram
```

Cycle Delays

Cycle delays can be used to postpone or delay the execution by a specified number of clock cycles or clocking events. The term cycle refers to the clock associated with the default clocking block. The **##** operator is used to specify cycle delay.

Syntax:

```
## integral_number | integer | (expression)
```

expression

Any SystemVerilog expression that evaluates to a positive integer value.

For example:

```
## 2;  
## (x+1);
```

Note:

VCS issues a compilation error if you use a cycle delay without specifying a default clocking block for the current module, interface, or program.

Input Sampling

All inputs and inouts of a clocking block are sampled at the *clocking_event* for that block. The following is skew related behavior involving regions:

- When the skew is #0, the signal value in the Observed region corresponds to the value sampled.
- When the skew is *not* #0, then the signal value at the Postponed region of the timestep skew time-units prior to the clocking event corresponds to the value sampled.
- When the skew is #1step, the signal value in the Preponed region corresponds to the value sampled.

The last sampled value of signal replaces the signal when the signal appears in an expression.

Note:

See section 14.3 of the SystemVerilog LRM 3.1a for definitions of Observed, Postponed and Preponed regions.

Synchronous Events

The event control operator, `@`, is used for explicit synchronization. This operator causes a process to wait for a particular event (that is, signal value change, or a clocking event) to occur.

Syntax

```
@ (expression);
```

expression

denotes clocking block input or inout, or a slice, which may include dynamic indices. The dynamic indices are evaluated when `@ (expression)` executes.

For examples, see pages 189-190 of the SystemVerilog LRM3.1a

Synchronous Drives

The output (or inout) signals defined in a clocking block are used to drive values onto their corresponding signals in the DUT at a specified time. That is, the corresponding signal changes value at the indicated clocking event as indicated by the output skew.

Note: For the syntax for specifying a synchronous drive, see section 15.14 of the SystemVerilog LRM 3.1a.

Consider the following clocking block and synchronous drives:

```
clocking cb1 @(posedge clk);
    default output #2;
    input #2 output #0 a = a1;
    output b = b1;
endclocking
```

```

initial
  begin
    @ (cb1); //synchronising with clocking event
    cb1.a <= 0; //drive at first posedge
    cb1.b <= 0; //drive after skew on first posedge
    ##2 cb1.a <= 1;
    ##1 cb1.b <= 1; //drive after 3 clock cycles
  end

```

The expression `cb1.a` (and `cb1.b`) is referred to as the *clockvar_expression* in the SystemVerilog LRM 3.1a (see page 190).

Note: Synchronous drives with a blocking cycle delay is supported.
 However, a synchronous drive with an intra cycle delay is not yet supported.

Drive Value Resolution

When the same net is an output from multiple clocking blocks, then the net is driven to its resolved signal value. When the same variable is an output from multiple clocking blocks, then the last drive determines the value of the variable.

Clocking Blocks in SystemVerilog Assertions

You can enter a clocking block as a clock signal for an assertion, property, or sequence. When you do the clocking event in the clocking block becomes the clock signal. The following is an example of a clocking block in an assertion:

```

clocking ck1 @(posedge clk) ;
:
endclocking

```

```
sequence seq ;
  @(ck1) a ##1 b ;
endsequence
```

```
A: assert property (@(ck1) a ##1 b) ;
N: assert property (seq) ;
```

In this example the clocking block named `ck1` is specified as the clock signal in the sequence and the two assertions. The clocking event in the clocking block, `posedge clk`, becomes the clock signal.

Sequences and Properties in Clocking Blocks

You can enter sequences and properties in a clocking block and then use them in an assertion. When you do, the clocking event for the clocking block becomes the clock signal for the sequence or property. The following is an example of a property in a clocking block:

```
clocking ck1 @(posedge clk) ;
property prop1;
  a ##1 b ;
endproperty
endclocking

A: assert property (ck1.prop1) ;
```

Here property `prop1` is declared in the clocking block named `ck1`. The clock signal for the property is `posedge clk`. You enter the property in an assertion by specifying the clocking block and then the property.

SystemVerilog Assertions Expect Statements

SystemVerilog assertions `expect` statements differ from `assert`, `assume`, and `cover` statements in the following ways:

- `expect` statements must appear in a SystemVerilog programs, whereas `assert`, `assume`, and `cover` statements can appear in modules, interfaces, and programs.
- You can declare sequences and properties and use them as building blocks for `assert`, `assume`, and `cover` statements, but this is not true for `expect` statements. `expect` statements don't have sequences, neither explicitly or implicitly. `expect` statements have properties, but properties are not explicitly declared with the `property` keyword.

Note:

- `assert`, `assume`, and `cover` statements cannot appear in clocking blocks.
- `assert`, `assume`, and `cover` statements can appear in programs on an early availability basis.

In an `expect` statement you specify a clock signal and have the option of specifying an edge for clocking events and delays, just like `assert` and `cover` statements, but these are not followed by a sequence, instead there is just a clock delay and an expression. There are action blocks that execute based on the truth or falsity of the expression. The clock delay can be a range of clocking events, and VCS evaluates the expression throughout that range. You can specify that the clock delay and evaluation of the expression must repeat a number of times (you can't both have a range of clocking events and also use repetition).

The following is an example of an `expect` statement:

```
e1: expect (@(posedge clk) ##1 in1 && in2)
    begin
        .          // statements VCS executes
        .          // if in1 && in2 is true
        .
    end
else
    begin
        .          // Statements VCS executes
        .          // if in1 && in2 is false
        .
    end
```

Where:

`e1:`

Is an instance name for the `expect` statement. You can use any unique name you want, followed by a colon (:).

`expect`

The `expect` keyword.

`(@(posedge clk) ##1 in1 && in2)`

Is the property of the `expect` statement. Such properties are enclosed in parentheses. This property is the following:

`@(posedge clk)`

the clock signal is `clk`, the clocking event is a rising edge (`posedge`) on `clk`. Using the `posedge` keyword means that it, with the clock signal, are an expression and so are also enclosed in parentheses.

`##1`

Is a clock delay. It specifies waiting for one clocking event, then evaluating the expression.

`in1 && in2`

Is an expression. If true, VCS executes the first action block

called the success block. if false VCS executes the second action block after the keyword `else`, called the failure block.

Here is another example of an expect statement. This one calls for evaluating the expression after a range of clocking events.

```
e2: expect (@(posedge clk) ##[1:9] in1 && in2)
    begin
        .          // statements VCS executes
        .          // if in1 && in2 is true
        .
    end
else
    begin
        .          // Statements VCS executes
        .          // if in1 && in2 is false
        .
    end
```

This expression calls for evaluation the expression after 1, 2, 3, 4, 5, 6, 7, 8, and 9 clocking events, a range of clocking events from 1 to 9.

Here is another example of an expect statement. This one calls for evaluating the expression to be true a number of times after the clock delay.

```
e3: expect (@(posedge clk) ##1 in1 && in2 [*5])
    begin
        .          // statements VCS executes
        .          // if in1 && in2 is true
        .
    end
else
    begin
        .          // Statements VCS executes
        .          // if in1 && in2 is false
        .
    end
```

[*] is the consecutive repeat operator. This expect statement calls for waiting a clock delay and then seeing if the expression is true, and doing both of these things five times in a row.

Note:

You can use the [*] consecutive repeat operator when you specify a range of clocking events such as ##[1:9].

The following is a code example that uses expect statements:

```
module test;
logic log1,log2,clk;

initial
begin
log1=0;
log2=0;
clk=0;
#33 log1=1;
#27 log2=1;
#120 $finish;
end

always
#5 clk=~clk;

tbpb tbpb1(log1, log2, clk);
endmodule

program tbpb (input in1, input in2, input clk);
bit bit1;

initial
begin

e1: expect (@(posedge clk) ##1 in1 && in2)
begin
bit1=1;
$display("success at %0t in %m\n", $time);
end
end
end
```

```

        else
            begin
                bit1=0;
                $display("failure at %0t in %m\n", $time);
            end
e2: expect (@(posedge clk) ##[1:9] in1 && in2)
            begin
                bit1=1;
                $display("success at %0t in %m\n", $time);
            end
        else
            begin
                bit1=0;
                $display("failure at %0t in %m\n", $time);
            end
e3: expect (@(posedge clk) ##1 in1 && in2 [*5])
            begin
                bit1=1;
                $display("success at %0t in %m\n", $time);
            end
        else
            begin
                bit1=0;
                $display("failure at %0t in %m\n", $time);
            end

end
endprogram

```

The program block includes an elementary clocking block, specifying a clocking event on the rising edge of `clk`, and no skew for signals `in1` and `in2`.

The `$display` system tasks in the failure and success action blocks display the following:

```
failure at 15 in test.tbpb1.e1
```

```
success at 65 in test.tbpb1.e2
```

success at 125 in test.tbpb1.e3

Virtual Interfaces

A Virtual Interface (VI) allows a variable to be a *dynamic reference* to an actual instance of an interface. VCS classifies such a variable with the Virtual Interface data type. Here is an example:

```
interface SBus;
    logic req, grant;
endinterface

module m;
    SBus sbus1();
    SBus sbus2();
    .
    .
    .
endmodule

program P;
virtual SBus bus;

initial
begin
    bus = m.sbus1; // setting the reference to a real
                  // instance of Sbus
    $display(bus.grant); // displaying m.sbus1.grant
    bus.req <= 1; // setting m.sbus1.req to 1
    #1 bus = m.sbus2;
    bus.req <= 1;
end
endprogram
```

Scope of Support

VCS supports virtual interface declarations in the following locations:

- program blocks and classes (including any named sub-scope)
- tasks or functions inside program blocks or classes (including any named sub-scope).

Variables with the virtual interface data type can be either of the following:

- SystemVerilog class members.
- Program, task, or function arguments.

You cannot declare a virtual interface in a module or interface definition.

Virtual Interface Modports

If only a subset of Interface data members is bundled into a modport, a variable can be declared as “virtual Interface_name.modport_name”:

Example.

```
interface SBus;
    logic req, grant;
    modport REQ(input req);
endinterface
program P;
virtual Sbus.REQ sb;
```

The semantic meaning is the same as in the example above with the difference that sb is now a reference only to a portion of Sbus and

writing assignments are subject to modport direction enforcement so that, for example, "sb.req = 1" would become illegal now (violates input direction of the modport REQ).

Driving a Net Using a Virtual Interface

There are two ways of driving interface nets from a testbench. These are:

- Via clocking block:

```
interface intf(input bit clk);
  wire w1;
  clocking cb @ (posedge clk);
    output w1;
  endclocking
endinterface
```

- By including a driver updated by a continuous assignment from a variable within the interface

```
interface intf;
  wire w1;
  reg r1;
  assign w1 = r1;
endinterface
```

This example demonstrates driving an interface net in a design.

Virtual Interface Modports and Clocking Blocks

You can reference an interface clocking block signal directly by using dot notation. The clocking information can be sent to the testbench only through modport if interface is having modports. The following example demonstrates this.

```

interface intf(input clk);
    int d;
    clocking cb @(posedge clk);
        default input #2 output #2;
        inout d;
    endclocking

    modport CB(clocking cb); //synchronous testbench modport
    modport master(inout d); //asynchronous testbench modport
endinterface

module top;
    bit clk;

    always #5 clk = ~clk;
    intf INTF(clk);
    tb TB(INTF);
endmodule

program tb(intf.CB ckb); //CB type modport is used to pass
    //clocking information with interface signals

    virtual intf.CB x = ckb;

    initial
        #200 $finish;

    initial begin
        x.cb.d <= 1; @(x.cb.d); $display($time,,x.cb.d);
        x.cb.d <= 2; @(x.cb.d); $display($time,,x.cb.d);
        //x.d <= 3; illegal as signal not visible via
        //CB modport
    end
endprogram

```

The output of this example is:

```

15      1
25      2

```


In the above example, if the modport passed to the testbench is of asynchronous type “intf.master” then the program will be:

```
program tb(intf.master ckb);
    virtual intf.master x = ckb;

    initial begin
        x.d <= 1;  @(x.d); $display($time,,x.d);
        x.d <= 2;  @(x.d); $display($time,,x.d);
    end
endprogram
```

The output of this example is:

```
0      1
0      2
```

Since clocking information is not passed through modport, the values are driven irrespective of clock.

Array of Virtual Interface

You can declare an array of virtual interfaces like any other array. Arrays of virtual interfaces can have aggregate assignment with arrays of interface instance. You can pass the entire array of virtual interfaces or a part of it can be passed to procedural methods (including class constructor).

The following program illustrates how to use arrays of virtual interfaces:

```
interface intf;
    int k = 30;
endinterface
```

```

program p;
  virtual intf VI[3:0];

  initial begin
    VI = top.INTF;    //aggregate assignment
    $display(VI[0].k);
    t1(VI); //passing whole VI array to task
  end

  task t1(virtual intf vif[3:0]);
    $display(vif[0].k);
  endtask
endprogram

module top;
  intf INTF [3:0] ();
endmodule

```

The output of this program is:

```

30
30

```

Driving/sampling of interface signals through Virtual Interface array elements is possible. For example:

```

VI[0].k <= 10;    //valid drive
@ (VI[0].k);    //valid wait

```

Clocking Block

Similarly to modport, a clocking block inside interface instance can be referred to by a virtual variable:

```

interface SyncBus(input bit clk);
  wire w;
  clocking cb @(posedge clk);
  output w;
endclocking

```

```

endinterface
....
program P;
virtual SyncBus vi;
...
initial vi.cb.w <= 1;
...
endprogram

```

In this case the assignment executes in accordance with the clocking block semantics.

Event Expression/Structure

Consider SyncBus as defined in the section “Clocking Block” .

```

task wait_on_expr(virtual SyncBus vi1, virtual SyncBus vi2);
    @(posedge (vi1.a & vi2.a))
        $display(vi1.b, vi2.b);
endtask

```

There is a principal difference between Vera and SV in that the “@” operator can have an operand that is a complex expression. We support all event expression that involve virtual interface variables.

Structures inside an interface can also be referred to by means of a virtual interface.

Null Comparison

We support:

- Comparison of vi variable with NULL.
- Runtime error if uninitialized virtual interface is used

```
begin
  virtual I vi;
  vi.data <= 1;
end
```

- NULL assignment

```
virtual I vi = NULL;
```

Not Yet Implemented

- Named type that involves virtual interface
 - typedef struct {reg rl; virtual I ii} T;
 - typedef virtual I T;
- Comparison of vi variables

By definition, `vi1 == vi2` iff they refer to the same instance of an interface (or both NULL).

- VI variables defined in the design.
- Class member variable access through VI when is declared in parent interface.

Coverage

The VCS implementation of SystemVerilog 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.

VCS collects all the coverage data during simulation and generates a database. VCS provides a tool to read the database and generate text or HTML reports.

The covergroup Construct

The `covergroup` construct specifies the set of cover points of interest, crosses of these cover points and the clocking event that tells VCS when to sample these cover points during simulation.

A covergroup can be declared inside a module or the program block. When declared inside a module they generate a separate instance of the covergroup for each instance of the module created.

```
program prog;

bit clk = 0;

enum {red, blue, yellow} colors;
colors my_color;

covergroup cg1 @(posedge clk);
    cp1 : coverpoint my_color;
endgroup

cg1 cg1_1 = new;

initial
repeat (15)
    #5 clk = ~clk;

initial
begin
    #40 my_color = blue;
    #23 my_color = yellow;
end
```

```
endprogram
```

This program contains the following:

- The enumerated data type colors, with members named red, blue, and yellow (whose default values are 0, 1, and 2).
- A variable of type colors called my_color
- A covergroup named cg1 that specifies the following:
 - the clocking event that is the rising edge on signal clk.
 - the coverage point that is to monitor the values of the variable my_color. The identifier of the coverage point, for hierarchical name purposes, is cp1.
- an instantiation of covergroup cg1 using the `new` method.

A covergroup can be defined inside a class. Furthermore there can be multiple covergroups in a class.

The following is an example of declaring a covergroup inside a class.

```
program P;  
    class MyClass;  
        int m_a;  
        covergroup Cov @(posedge clk);  
            coverpoint m_a;  
        endgroup  
  
        function new();  
            Cov = new;  
        endfunction  
    endclass  
endprogram
```

Defining a Coverage Point

In a coverage point definition you can specify the following:

- bins for value ranges
- bins for value transitions
- bins for illegal coverage point values

Bins for Value Ranges

You use the curly braces { } and the `bins` keyword to specify the bins for a coverage point, for example:

```
covergroup cg1 @ (posedge clk);
coverpoint data
{
bins some_name [] = {[1:20]};
}
endgroup
```

In coverage point data:

- The keyword `bins` specifies one or more bins for coverage data.
- The name `some_name` is an identifier for all the bins. It is the root bin name.
- The empty square brackets `[]` specifies there will be a separate bin for each value in the specified range.
- The range of value follows the equal sign `=` and is in the nested set of curly braces `{ }`. This range is 1 through 20. The range is always specified as *lowest_value:highest_value*.

Coverage point data will have 20 bins, the first named some_name_1 and the last named some_name_20.

Bin Value/ Range Resolution

The values corresponding to the range specified by a bin for a coverpoint are resolved according to the precision and sign-ness of the corresponding coverpoint expression.

To evaluate a bin hit and update the hit count the coverpoint and bin expressions are compared and evaluated.

A warning is generated in the following cases.

- If the coverpoint expression is unsigned and bin expressions is signed and is a negative value.
- If the assignment equivalence of the coverpoint and bin expression does not match.
- If the bin expression evaluates to a value that contains X or Z bits.

The following rules shall apply when a warning is issued for a bin element:

- If there is a single element described in the bin range which is outside the scope of the evaluated coverpoint expression then the element is not considered for the bin expression.
- If the elements that describe the bin range, contains x or z bits then every value in the range would generate a warning and the elements are not considered for the bin expression.
- If the elements that describe the bin range are outside the scope of the evaluated coverpoint expression then the bin range is accordingly adjusted to the min and max values corresponding to the evaluated coverpoint values.

Example :

```
bit [1:0] cp_exp1;
    // type evaluates to the value range of 0 to 3
bit signed [2:0] cp_exp2;
    // type evaluates to the value range of -4 to 3
covergroup g1 @(posedge clk);
coverpoint cp_exp1 {
    bins b_exp1 = {1, [2:5] };
    bins b_exp2 = { -1, [1:4]};
}
coverpoint cp_exp2 {
    bins b_exp3 = {1, [2:5], [6:10] };
}
endgroup
```

Warnings Issued and their resolutions

A warning is issued for the bin `b_exp1` since the range `[2:5]` exceeds the upper bound for the coverpoint `cp_exp1`. The bin `b_exp1` is evaluated and adjusted to the max value of the coverpoint. Here the bin `b_exp1` is evaluated as `{1, [2:3]}`.

A warning is issued for the bin `b_exp2` since the singleton value `-1` exceeds the upper bound for the coverpoint `cp_exp1`. The bin `b_exp2` is evaluated such that the `-1` value is not considered. Here the bin `b_exp2` is evaluated as `{[1:3]}`.

A warning is issued for the bin `b_exp3` since the range `[2:5]` and `[6:10]` exceeds the upper bound for the coverpoint `cp_exp3`. The bin `b_exp3` is evaluated such that the range `[2:5]` is adjusted to the max value of the coverpoint and the range `[6:10]` is completely ignored. Here the bin `b_exp3` is evaluated as `{1, [2:3]}`.

You can specify different bins for different value ranges, for example:

```
coverpoint data
{
    bins first_ten = {[1:10]};
    bins second_ten = {[11:20]};
}
```

Here the coverage information about when the coverage point data has the values 1 to 10 is in the bin named `first_ten`, and the information about when data has the values from 11 to 20 is in the bin named `second_ten`.

You can specify a default bin with the `default` keyword, for example:

```
coverpoint data
{
    bins bin1 = {[1:5]};
    bins bin2 = {[6:10]};
    bins bin3 = default;
}
```

In this example coverage information about when data has the values 1-10 is in bins `bin1` and `bin2`, information about all other values is in `bin3`.

You can specify a list of value ranges for example:

```
coverpoint data
{
    bins bin1 = {[0:3], 5, 7, [9:10]};
    bins bin2 = {4, 6, 8};
    bins bin3 = default;
}
```

Here the information about when data is 0, 1, 2, 3, 5, 7, 9, and 10 is in bin1, the information about when data is 4, 6, and 8 is in bin2, and the information about when data has any other value is in bin3.

When you instantiate the covergroup, you can make the covergroup a generic covergroup and then pass integers to specify value ranges in the `new` method, for example:

```
covergroup cg1 (int low, int high) @ (posedge clk);
    coverpoint data
    {
        bins bin1 = {[low:high]};
    }
endgroup

cg1 cg1_1 = new(0,10); // 0 is the low value
                       // 10 is the high value
                       // of the range
```

Bins for Value Transitions

In a transition bin you can specify a list of sequences for the coverpoint. Each sequence is a set of value transitions, for example:

```
coverpoint data
{
    bins from0 = (0=>1), (0=>2), (0=>3);
    bins tran1234 = (1=>2=>3=>4);
    bins bindef = default;
}
```

In this example, coverage information for the sequences 0 to 1, 0 to 2, or 0 to 3 is in the bin named `from0`. Coverage information about when data transitions occurred from 1 to 2 and then 3 and then 4 is in bin `tran1234`.

You can use range lists to specify more than one starting value and more than one ending value, for example:

```
bins from1and5to6and7 = (1,5=>6,7);
```

Is the equivalent of:

```
bins from1and5to6and7 = (1=>6, 1=>7, 5=>6, 5=>7);
```

You can use the repetition [*] operator, for example:

```
bin fivetimes3 = (3 [*5]);
```

Is the equivalent of:

```
bin fivetimes3 = (3=>3=>3=>3=>3);
```

You can specify a range of repetitions, for example:

```
bin fivetimes4 = (4 [*3:5]);
```

Is the equivalent of:

```
bin threetofivetimes4 = (4=>4=>4,4=>4=>4=>4,4=>4=>4=>4=>4);
```

Specifying Illegal Coverage Point Values

Instead of specifying a bin with the `bins` keyword, use the `illegal_bins` keyword to specify values or transitions that are illegal.

```
coverpoint data
{
    illegal_bins badvals = {7,11,13};
    illegal_bins badtrans = (5=>6,6=>5);
    bins bindef = default;
```

```
}
```

VCS displays an error message when the coverage point reaches these values or makes these transitions.

Defining Cross Coverage

Cross coverage is when there are two coverage points that you want VCS to compare to see if all the possible combinations of the possible values of the two coverage points occurred during simulation.

Consider the following example:

```
program prog;
bit clk;
bit [1:0] bit1,bit2;

covergroup cg1 @(posedge clk);
  bit1: coverpoint bit1;
  bit2: coverpoint bit2;
  bit1Xbit2: cross bit1, bit2;
endgroup

cg1 cg1_1 = new;

initial
begin
  clk = 0;
  repeat (200)
  begin
    bit1 = $random();
    bit2 = $random();
    #5 clk = ~clk;
    #5 clk = ~clk;
  end
end

endprogram
```

In covergroup cg1 there are two coverpoints labeled bit1 and bit2. In addition, there is the following:

1. the `bit1Xbit2` identifier for the cross.
2. The `cross` keyword, specifying the coverpoints to be crossed.

Both coverpoints are two-bit signals. The four possible values of each are 0 through 3. There are 16 possible combinations of values.

The `prog.txt` file for this code contains the following:

```
Automatically Generated Cross Bins
```

bit1	bit2	# hits	at least
auto[0]	auto[0]	10	1
auto[0]	auto[1]	13	1
auto[0]	auto[2]	12	1
auto[0]	auto[3]	5	1
auto[1]	auto[0]	12	1
auto[1]	auto[1]	18	1
auto[1]	auto[2]	10	1
auto[1]	auto[3]	13	1
auto[2]	auto[0]	19	1
auto[2]	auto[1]	16	1
auto[2]	auto[2]	17	1
auto[2]	auto[3]	6	1
auto[3]	auto[0]	6	1
auto[3]	auto[1]	15	1
auto[3]	auto[2]	16	1
auto[3]	auto[3]	12	1

There are 16 cross coverage bins, one for each possible combination.

Defining Cross Coverage Bins

The `binsof` construct supplies the coverage bins for the expression argument, which can be either a coverpoint or a coverpoint bin.

```
covergroup cg1 @(posedge clk);
  cp1 : coverpoint bit1
  {
    bins lowcp1vals = {[0:7]};
    bins hicp1vals = {[8:15]};
  }
  cp2 : coverpoint bit2
  {
    bins lowcp2vals = {[0:7]};
    bins hicp2vals = {[8:15]};
  }
  cp1Xcp2 : cross cp1, cp2
  {
    bins bin1 = binsof(cp1) intersect {[0:7]};
    bins bin2 = binsof(cp1.hicp1vals) ||
                binsof(cp2.hicp2vals);
    bins bin3 = binsof(cp1) intersect {[0:1]} &&
                binsof(cp2) intersect {[0:3]};
  }
endgroup
```

In this example, the respective cross coverage bins, `bin1`, `bin2`, and `bin3` receive data whenever the corresponding right hand side `binsof` expressions are satisfied. For example, `bin1` receives data when any bin of `cp1` whose value range overlaps with the range `[0:7]` receives data. In this case `bin1` receive data whenever bin `lowcp1vals` of coverpoint `cp1` receives data.

Similarly, cross bin, `bin2` receives data whenever either bin `hicp1vals`, of coverpoint `cp1`, receives data, or bin `hicp2vals`, of cover point `cp2`, receives data. Cross bin, `bin3` receives data if any bin of cover point `cp1`, whose value range overlaps with the range `[0:1]`, receives data, and, for the same sample event occurrence, any bin of cover point `cp2`, whose value range overlaps with the range `[0:3]`, also receives data. In this example, cross bin `bin3` receives data when bin `lowcp1vals`, of cover point `cp1`, receives data, and bin `lowcp2vals`, of cover point `cp2`, also receives data.

If none of the user-defined cross bins match, then VCS automatically creates an auto cross bin to store the hit count for each unique combination of the cover point bins.

Cumulative and Instance-based Coverage

Coverage statistics can be gathered both cumulatively and on a per-instance basis.

Cumulative Coverage

Cumulative implies that coverage statistics (that is, bin hit counts and coverage numbers) are computed for the covergroup definition. In this case all instances of the covergroup contribute to a single set of statistics maintained for the covergroup definition. By default, VCS computes cumulative coverage information.

An example of when this kind of coverage is useful is when covering a packet class. Cumulative coverage will provide information for all the packet instances of the class.

Note:

In cumulative mode, only cumulative information can be queried for. Furthermore, the coverage reports only report on cumulative data for the covergroup definitions, and not instances.

Instance-based Coverage

Instance-based coverage statistics involve computing coverage statistics for every instance of a covergroup as well as the covergroup definition as a whole. VCS computes per-instance coverage statistics when you set the cumulative attribute of the option.per_instance to 1.

Module based Covergroup Instances

The covergroups defined inside a module creates a separate instances of the covergroup for every instance of the module created. The coverage statistics is calculated for every instance of the module created separately even when the option.per_instance is set to 0.

Coverage Options

You can specify options for the coverage of a covergroup with the `type_option.option=argument` keyword and argument for specifying options, for example:

```
covergroup cg2 @(negedge clk);
type_option.weight = 3;
type_option.goal = 99;
type_option.comment = "Comment for cg2";
  cp3 : coverpoint bit3;
  cp4 : coverpoint bit4;
endgroup
```

These options specify the following:

```
type_option.weight = integer;
```

Specifies the weight of the covergroup when calculating the overall coverage. Specify an unsigned integer. The default weight value is 1.

```
type_option.goal = integer;
```

Specifies the target goal of the covergroup. Specify an integer between 0 and 100. The default goal is 90.

Note:

“Coverage number” is a percentage. If all bins of a covergroup are covered, then the coverage number for that covergroup is 100%.

```
type_option.comment = "string";
```

A comment in the report on the covergroup.

You can also apply these option to coverage points, for example:

```
covergroup cg1 @(posedge clk);
  cp1 : coverpoint bit1
  {
    type_option.weight = 333;
    type_option.goal = 50;
    type_option.comment = "Comment for bit1";
  }
  cp2 : coverpoint bit2;
endgroup
```

You can also apply these options to instances using the `instance_name.option.option_name=argument` keyword and argument, for example:

```
covergroup cg1 @(posedge clk);
  cp1 : coverpoint bit1;
  cp2 : coverpoint bit2;
endgroup
```

```

cgl cgl_1 = new;

initial
begin
    cgl_1.option.weight = 10;
    cgl_1.option.goal = 75;
    .
    .
    .
end

```

Instance specific options are procedural statements in an initial block.

There are additional options that are just for instances:

instance_name.option.at_least=integer

Specifies the minimum number of hits in a bin for VCS to consider the bin covered.

instance_name.option.detect_overlap=boolean

The *boolean* argument is 1 or 0. When *boolean* is 1, VCS displays a warning message when there is an overlap between the range list or transitions list of two bins for the coverage point.

instance_name.option.name [=string]

This option is used to specify a name for the covergroup instance. If a name is not specified, a name is automatically generated.

instance_name.option.per_instance=boolean

The *boolean* argument is 1 or 0. When *boolean* is 1, VCS keeps track of coverage data for the instance.

Predefined Coverage Methods

SystemVerilog provides a set of predefined covergroup methods described in this section. These predefined methods can be invoked on an instance of a covergroup. They follow the same syntax as invoking class functions and tasks on an object.

Predefined Coverage Group Functions

The predefined methods supported at this time are:

- `get_coverage()`
- `get_inst_coverage()`
- `set_inst_name(string)`
- `sample()`
- `stop()`
- `start()`

get_coverage()

Calculates the coverage number for the covergroup type (see page 228 for definition). Return type: real.

Below is an example of using `get_coverage()` to calculate the coverage number of a covergroup:

```
program test();
reg clk = 0;
reg [2:0] var = 3'b001;
class A;
covergroup covType @(clk); //covergroup, covType, defined
                        //in class A,
    cp1: coverpoint var {
```

```

        bins s0 = {[ 0 : 2]} ;
        bins s1 = { 3 };
        bins s2 = { 4 };
        bins s3 = { 5 };
        bins s4 = { 6 };
        bins s5 = { 7 };
    }
endgroup

function new;
    covType = new(); //instantiate the embedded covergroup
endfunction

endclass

A A_inst;

initial begin
    repeat (10) begin
        #5 clk = ~clk;
        var = var + 1;
    /* get_coverage() calculates the number of the embedded
       covergroup covType as a whole */
        $display("var=%b coverage=%f\n", var,
            A_inst.covType.get_coverage());
    end
end

initial
    A_inst = new();

endprogram
Output of program:
var=010 coverage=0.000000

var=011 coverage=16.666666

var=100 coverage=33.333332

var=101 coverage=50.000000

var=110 coverage=66.666664

```

```
var=111 coverage=83.333336
var=000 coverage=100.000000
var=001 coverage=100.000000
var=010 coverage=100.000000
var=011 coverage=100.000000
```

See the [get_coverage\(\), stop\(\), start\(\) example](#) on [page 235](#) for another example of using the `get_coverage()` function.

get_inst_coverage()

Calculates the coverage number for coverage information related to the covergroup instance. Return type: real.

```
program test();
reg clk = 0;
reg [2:0] var = 3'b001;

covergroup covType (input integer param1) @(clk);
  cp1: coverpoint var {
    bins s0 = { [ 0 : param1] } ;
    bins s1 = { 3 };
    bins s2 = { 4 };
    bins s3 = { 5 };
  }
type_option.per_instance =1;
endgroup

covType cov1;

initial begin
  repeat (5) begin
    #5 clk = ~clk;
    var = var + 1;
    $display("var=%b coverage=%f\n", var,
             cov1.get_inst_coverage());
  end
end
```

```

        end
    end

    initial
        cov1 = new(2);

endprogram

```

The output of the program is:

```

valr=010 coverage=0.000000
valr=011 coverage=25.000000
valr=100 coverage=50.000000
valr=101 coverage=75.000000
valr=110 coverage=100.000000

```

set_inst_name(*string*)

The instance name is set to *string*. Return type: void.

In the example below, `cov1` is the name of an instance that is of type `covType`, declared as such (`covType cov1; cov1 = new(2);`). The name is changed to `new_cov1` after calling `set_inst_name("new_cov1")`.

```

program test();
    reg clk = 0;
    reg [2:0] var = 3'b001;

    covergroup covType (input integer param1) @(clk);
        cp1: coverpoint var {
            bins s0 = { [ 0 : param1] } ;
            bins s1 = { 3 };
        }
    endgroup

    covType cov1;

```

```

initial
begin
    cov1 = new(2);
    $display("Original instance name is %s\n",
            cov1.option.name);
    cov1.set_inst_name("new_cov1");//change instance name
    $display("Instance name after calling set_inst_name()
            is %s\n", cov1.option.name);
end

endprogram

```

Output of the program is:

Original instance name is cov1

Instance name after calling set_inst_name() is new_cov1

sample()

sample() triggers sampling of the covergroup instance. Return void.

```

program test();
reg clk = 0;
reg [2:0] var = 3'b001;

covergroup covType ();
    cpl: coverpoint var {
        bins s0 = { 0 };
        bins s1 = { 1 };
        bins s2 = { 2 };
        bins s3 = { 3 };
        bins s4 = { 4 };
        bins s5 = { 5 };
        bins s6 = { 6 };
        bins s7 = { 7 };
    }
endgroup

```



```

covType cov1;

initial
    cov1 = new();

initial begin
    repeat (10) begin
        #10 clk = ~clk;
        var = var + 1;
    end
end
initial begin
    repeat (40) begin
        #3  cov1.sample();
    end
end
endprogram

```

stop()

When called, collecting of coverage information is stopped for that instance. Return type: void. See the [get_coverage\(\), stop\(\), start\(\) example](#) on [page 235](#).

start()

When called, collecting of coverage information resumes for that instance. Return type: void. See the [get_coverage\(\), stop\(\), start\(\) example](#) on [page 235](#).

get_coverage(), stop(), start() example

```

program test();
reg clk = 0;
reg [2:0] var = 3'b001;

covergroup covType (input integer param1) @(clk);
    cp1: coverpoint var {
        bins s0 = { [ 0 : param1]  } ;
        bins s1 = { 3  };
    }
endcovergroup

```

```

        bins s2 = { 4 };
        bins s3 = { 5 };
        bins s4 = { 6 };
        bins s5 = { 7 };
    }
endgroup

covType cov1;

initial begin
    repeat (10) begin
        #5 clk = ~clk;
        var = var + 1;
        $display("var=%b covergae=%f\n", var,
                cov1.get_coverage());
    end
end

initial
begin
    cov1 = new(2);
    cov1.stop();
    cov1.option.weight = 5;
    #30 cov1.start();
end

endprogram

```

OUTPUT:

```

var=010 covergae=0.000000
var=011 covergae=0.000000
var=100 covergae=0.000000
var=101 covergae=0.000000
var=110 covergae=0.000000

```

```
var=111 covergae=0.000000
var=000 covergae=16.666666
var=001 covergae=33.333332
var=010 covergae=33.333332
var=011 covergae=33.333332
```

Unified Coverage Reporting

The db based coverage reporting has been replaced by the Unified Report Generator (URG). The URG generates combined reports for all types of coverage information. The reports may be viewed through through an overall summary "dashboard" for the entire design/testbench.

The format of the text report generated by the URG is enhanced and different from the text report that used to be generated by the `vcs -cov_report` command line options. Any scripts that use these old command line options now need to be modified to use the URG options.

The functional coverage database files and their location have been changed. The coverage database is written to a top-level coverage directory. By default this directory name is `simv.vdb`. In general its name comes from the name of the executable file, with the `.vdb` extension. The reporting tool shipped with VCS version 2006.06 cannot read coverage databases generated using previous versions of VCS. Similarly, the reporting tool shipped with VCS 2006.06 versions cannot read coverage databases generated using VCS 2006.06.

The Coverage Report

During simulation VCS creates a default database directory `simv.vdb`. For the code example above, VCS writes the `simv.vdb` directory in the current directory. This directory contains the coverage database and all the information VCS needs to write a coverage report. There are two types of coverage reports:

- an ASCII text file
- an HTML file

The ASCII Text File

The command to instruct VCS to generate a coverage report in ASCII format is:

```
urg -dir simv.vdb -format text
```

The `-format text` option instructs VCS to generate, text reports (`dashboard.txt`, `grpinfo.txt`, `tests.txt` and `groups.txt`) in a separate directory called `urgReport`. The ASCII coverage report `grpinfo.txt` for the previous code example is as follows:

```
Group : cg1_SHAPE_0
=====
Group : cg1_SHAPE_0
=====
Score   Weight  Goal
100.00  1         100
-----

Samples for Group : cg1_SHAPE_0

Variable   Expected   Covered   Percent   Goal   Weight
```

Total	3	3	100.00		
cp1	3	3	100.00	100	1

Summary for variable cp1

	Expected	Covered	Percent
User Defined Bins	3	3	100.00

User Defined Bins for cp1

Bins name	count	at least
auto_yellow	2	1
auto_blue	2	1
auto_red	4	1

The report begins with a summary of the coverage for the covershapes created. There is 100% coverage, meaning that VCS saw all possible values for the coverage point(s) in the covergroup.

For coverage point cp1, the report says the following:

- The coverage point, in this case variable my_color, reached all its possible values.
- There were no user defined bins.
- VCS created bins for the coverage point named auto_blue, auto_red, and auto_yellow, all named after the members of the enumerated type colors, blue, red, and yellow.
- There were four hits for bin auto_red, this means that during simulation, there were four clocking events (rising edge on signal clk) where VCS detected that the value of the variable my_color was red.
- There were two hits for bins auto_blue and auto_yellow.

The HTML File

The command to instruct VCS to generate a coverage report in HTML format is:

```
urg -dir simv.vdb
```

This command instructs VCS to generate coverage reports (dashboard.html, grp0.html, tests.html and groups.html) in the directory urgReports. The HTML coverage report grp0.html for the previous code example is as follows:

Group : cg1_SHAPE_0

[dashboard](#) | [hierarchy](#) | [modlist](#) | [groups](#) | [tests](#) | [asserts](#)

Group : cg1_SHAPE_0

Score	Weight	Goal
100.00	1	100

Samples for Group : cg1_SHAPE_0

Variable	Expected	Covered	Percent	Goal	Weight
Total	3	3	100.00		
cp1	3	3	100.00	100	1

Go to [top](#)

Summary for variable cp1

	Expected	Covered	Percent
User Defined Bins	3	3	100.00

User Defined Bins for cp1

Bins

name	count	at least
auto_yellow	2	1
auto_blue	2	1
auto_red	4	1

Go to [top](#)

[dashboard](#) | [hierarchy](#) | [modlist](#) | [groups](#) | [tests](#) | [asserts](#)

The report has links to other .html reports. Click on them to see the appropriate information.

Please refer to the [Unified Report generator](#) for more details

Persistent Storage of Coverage Data and Post-Processing Tools

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.

```
$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 24-5 Unique Name Generation Scheme

Test Name	Database
pci_test	Database for the first testbench run.
pci_test_gen_1	Database for the second testbench run
pci_test_gen_2	Database for the 3rd testbench run
pci_test_gen_n	Database for the nth 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_file (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 the Example 1-1, below, there is a SystemVerilog class `MyClass` with an embedded covergroup `covType`. VCS finds the cumulative coverage data for the covergroup `MyClass:covType` in the database file `Run1` and loads it into the `covType` embedded coverage group in `MyClass`.

Example 24-45

```
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 commands above direct VCS to find the coverage data for the specified instance name in the database, and load it into the coverage instance.

In the Example 1-2, 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 24-46

```
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 over write the calls to `coverage_set_test_database_name` and loading 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, VCS automatically adds the extension `.vdb` to the directory name if not specified.

```
-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".

VCS NTB (SV) Memory Profiler

The VCS NTB(SV) memory profiler is a Limited Customer availability (LCA) feature in NTB(SV) and requires a separate license. Please contact your Synopsys AC for a license key.

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

Use Model

The `$vcsmemprof()` task can be called from the CLI or the UCLI interface. The syntax for `$vcsmemprof()` is as follows:

```
$vcsmemprof("filename", "w|a");
```

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 -ntb -sverilog +dmprof dut_filename.v testbench_filename.sv \  
    [-debug | -debug_all]
```

Note:

Use the `-sverilog` compile-time option when compiling SystemVerilog code.

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+")}
```

CLI Interface

Compile-time

The dynamic memory profiler is enabled only if you specify `+dmprof` on the VCS compile-time command line:

```
vcs -ntb -sverilog +dmpref dut_filename.v testbench_filename.sv \  
[-debug | -debug_all]
```

Note:

Use the `-sverilog` compile-time option when compiling SystemVerilog code.

Runtime

At runtime, invoke `$vcsmemprof()` from the CLI command line prompt as follows. You can make the call to `$vcsmemprof()` at any point during the simulation:

```
simv -s //Invokes the cli prompt  
cli_0>$vcsmemprof("memprof.txt", "w|a+")
```

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.

Incremental Profiling

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 issue.

Only Active Memory Reported

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.

VCS NTB (SV) Dynamic Memory Profile Report

The profile report includes the following sections.

1. Top level view

Reports the total dynamic memory consumed in all the SV programs and that consumed in all the SV 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 SV program in the system.

4. Program To Construct View

a. Task-Function-Thread section

Reports the total dynamic memory in each active task, function and thread in the module/program.

b. Class Section

Reports the total dynamic memory consumed in each class in the module/program.

c. Dynamic data Section

Reports the total memory consumed in each of 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".

The Direct Programming Interface (DPI)

The Direct Programming Interface (DPI) is a Limited Customer availability (LCA) feature in NTB (SV) and requires a separate license. Please contact your Synopsys AC for a license key.

The DPI is an interface between SystemVerilog and another language such as C. Using DPI you can directly call a function in the other language.

Note:

Currently DPI is implemented only for the C/C++ languages.

Import Functions - SystemVerilog calling C/C++

The import task/functions are C/C++ functions that can be called from the SystemVerilog code. You must declare them before you can use them. After you declare an import function, you can call it like any procedural statement in your SystemVerilog code.

Import tasks and functions can have 0 or more formal input, output, and inout arguments. Import functions can either return a value or be defined as void, while import tasks never return a value.

Import functions can have the properties context or pure while the import tasks can have the property context only.

Syntax:

```
Declaration :  
import "DPI" [cname =] [pure] function type name (args);  
import "DPI" [cname =] [pure][context] task name (args);
```

Example:

```
import "DPI" context task c_task(input int addr);

program top;
  initial c_task(1000);
  initial c_task(2000);
endprogram

#include <svdpi.h>
void c_task(int addr) {
  ...
}

vcs -sverilog top.sv c_task.c
```

Export Functions- C/C++ Calling SystemVerilog

The export tasks/functions are SystemVerilog tasks/functions that can be called from C/C++ languages. You must declare them before you can use them. Export function declarations are allowed only in the scope in which the function is defined and only one declaration per function is allowed in a given scope.

The export functions and tasks have the same restrictions on argument types and return values as the import functions.

Syntax:

```
Declaration :
export "DPI" [cname =] function name (args);
export "DPI" [cname =] task name (args);
```

Example:

```
import "DPI" task c_test(int addr);
initial c_test(1000);
export "DPI" task sv_add;
function int sv_add(input int a, input int b);
    sv_add = a+b;
endtask

#include <svdpi.h>
extern void sv_add(int, int);
void c_test(int addr) {
    ...
    sv_add(addr, 10);
    ...
}

vcs -sverilog top.sv c_test.c
```

Note:

You can export all SystemVerilog functions except class member functions.

Limitations

In the current implementation of the DPI, import and export functions are supported with some restrictions on the following data types:

- Unpacked structs/unions, byte, shortint, longint, and shortreal.
- Unpacked arrays of the above types

Include Files

The SystemVerilog 3.1a LRM defines the C layer for the DPI. The C-layer of the DPI provides two include files:

- The main include file, `$VCS_HOME/lib/svdpi.h`, is defined in the SystemVerilog 3.1 standard and defines the canonical representation, all basic types, and all interface functions. The file also provides function headers and defines a number of helper macros and constants. See section D.9.1 of the SystemVerilog 3.1 LRM.

The following functions are not implemented:

<code>svGetNameFromScope</code>	<code>svAckDisabledState</code>
<code>svGetCallerInfo</code>	<code>svIsDisabledState</code>

- The second include file, `$VCS_HOME/lib/svdpi_src.h`, defines only the actual representation of packed arrays and its contents are unique to VCS.

Time Consuming Blocking Tasks

When SystemVerilog calls a C import task, this task can then call blocking (context) SystemVerilog tasks. This is particularly useful if the C code must call a read/write task in a SystemVerilog BFM model.

Note that the C task must be initially called from SystemVerilog using an import task call. For example, the following C code contains a test that calls the SystemVerilog `apb_write` task as needed to issue writes on an APB bus.

```
#include<svdpi.h>
extern void apb_write(int, int);

void c_test(int base) {
    int addr, data;
    for(addr=base; addr<base+5; addr++) {
        data = addr * 100;
        apb_write(addr, data);
        printf("C_TEST: APB_Write : addr = 0x%8x, data = 0x%8x\n",
addr, data);
    }
}
```

SystemVerilog File

This SV code calls `C_test()`, which then calls the blocking `APB_Write` task in SystemVerilog. The `APB_Write` task has a simple semaphore to ensure unique access.

```
import "DPI" context task c_test(input int base_addr);
program top;
    semaphore bus_sem = new(1);

    export "DPI" task apb_write;
    task apb_write(input int addr, data);
        bus_sem.get(1);
        #10 $display("VLOG : APB Write : Addr = %x, Data = %x",
", addr, data);
        #10 ifc.addr <= addr;
        #10 ifc.data <= data;
        bus_sem.put(1);
    endtask

    initial begin
        fork
            c_test(32'h1000);
            c_test(32'h2000);
        join
    end

endprogram
```

The VCS command line compiles the files.

```
vcs -sverilog top.sv c_test.c -R
```

25

Source Protection

Source protection changes the source description of a Verilog model so that others cannot read or understand it but they can still simulate it. Source protection enables you to protect proprietary information about your designs but enables users in companies that purchase your designs to simulate them.

VCS provides three ways to protect your source description:

- **Mangling of source description:**
Mangling replaces identifiers in the source description with identifiers that obscure the design. Mangling does not change the structure of the source description in that the keywords and syntax remains intact.
- **Encrypting of source description:**
Encrypting makes the entire source description unreadable. SDF files can also be encrypted, however, you cannot mangle or make a binary executable from SDF files.

- Creating a VMC model from source description:
A VMC model is an executable binary that you compile from your Verilog model. To simulate the VMC model you link it to VCS. VMC is a separate Synopsys product.

These source protection methods vary in the levels of security they provide, their impact on performance, platform and vendor independence, and PLI and CLI access as described in Table 25-1, Source Protection Methods.

Table 25-1 Source Protection Methods

Method	Level of Security	Performance	Platform Independence	Vendor Independence	PLI and CLI Access
Mangling	Alters only the identifiers. You can read the structure of the source description.	Same as VCS or third party Verilog simulator	Yes, the output is an ASCII file	Yes, you can simulate the resulting model in any Verilog simulator	Yes, however the identifiers are difficult to understand.
Encrypting	Alters the entire source description but some identifiers can be seen in generated C code	Same as VCS	Yes, the output is an ASCII file	No, the encrypted output is in a format that only VCS can read.	Yes, but only if you know the identifiers before they were encrypted
VMC	Absolute, the output is an executable binary	Some impact on small and active VMC models	No, you can only simulate a VMC model on the platform that was used to generate it.	Yes, using standard PLI access	No access into VMC models

Note:

This chapter describes mangling and encrypting. For information on creating a VMC model see the *VMC User's Guide*.

Encrypting Source Files

Encrypt Using Compiler Directives

You use compiler directives and command line options to encrypt source and SDF files. VCS creates copies of your files with encrypted contents. The following sections describe how you encrypt Verilog source and SDF files:

- The section “Encrypting Specified Regions” on page 25-4 describes how you can specify what part of your source description you want VCS to encrypt.
- The section “Encrypting The Entire Source Description” on page 25-5 describes how you can encrypt all the contents of your source description.
- The section “Encrypting SDF Files” on page 25-9 describes the source protection options for SDF files.
- The section “Specifying Encrypted Filename Extensions” on page 25-10 specifies how you can override the default filename extensions for encrypted files. The default extension is .vp.
- The section “Specifying Encrypted File Locations” on page 25-10 describes how you can specify the location of the encrypted files that VCS creates. The default is the same directory as the original source or SDF file.
- The section “Multiple Runs and Error Handling” on page 25-10 describes how you can circumvent problems that can occur when you perform multiple encryptions of the same files or encounter error conditions.

Encrypting Specified Regions

You can control what part of your source VCS encrypts in the new files and which part remains readable. To do so you do the following:

1. Enclose the part of your source that you want VCS to encrypt between the ``protect` and the ``endprotect` compiler directives.
2. Include the `+protect` option on the `vcs` command line.

When you run VCS with the `+protect` option:

- VCS creates new files for the Verilog source files you specify on the `vcs` command line.
- VCS replaces the ``protect` and ``endprotect` compiler directives with the ``protected` and ``endprotected` compiler directives and encrypts all the Verilog source description between these ``protected` and ``endprotected` compiler directives.

The new files have the same filename except that VCS appends a “p” to the filename extension of the new files. For example, if you run source protection for a source file named `my_design.v`, VCS names the new file with encrypted source `my_design.vp`. This is the default filename extension for the new file. You can specify a different extension.

The following is an example of a Verilog source file with a region of the source that you want protected:

```
module top( inp, outp);  
input [7:0] inp;  
output [7:0] outp;  
reg [7:0] count;  
assign outp = count;
```

```

always
begin:counter
`protect //begin protected region
reg [7:0] int;
count = 0;
int = inp;
while (int)
begin
if (int [0]) count = count + 1;
int = int >> 1;
end
`endprotect //end protected region
end
endmodule

```

The following is the contents of the new file after you run source protection with the +protect option:

```

module top( inp, outp);
input [7:0] inp;
output [7:0] outp;
reg [7:0] count;
assign outp = count;
always
begin:counter
`protected
%%AqDtf%,%,%,%,%,%,-%,%,UB@%,|%5%B%<z%,NIS%A-%,%,DH
?%,NIW%A-%,%,PONKB%,%4NIS%3-%,%,%,%,EB@NI-%,%,%,%,%
RIS%,%?%,DHRIS%,%1%,%;%A-%,%,%,%,%,%,NIS%,%?%,NIS%
,%, $
`endprotected
//end protected region
end
endmodule

```

Encrypting The Entire Source Description

You can encrypt all the modules and UDPs in the new Verilog source files that VCS creates without yourself entering ``protect` and ``endprotect` compiler directives in your source code.

To do so include the `+autoprotect`, `+auto2protect` or `+auto3protect` option on the `vcs` command line. The difference between these options is where VCS begins to encrypt the Verilog source description in modules and UDPs. This difference is as follows:

`+autoprotect`

The `+autoprotect` option encrypts the module port list (or UDP terminal list) in addition to the body of the module (or UDP.)

`+auto2protect`

The `+auto2protect` option does not encrypt the module port list (UDP terminal list). Instead encryption begins after the semicolon following the port or terminal list. This difference produces a syntactically correct module or UDP header statement.

`+auto3protect`

The `+auto3protect` option does what the `+auto2protect` option does except that it does not encrypt parameter declarations that precede the first port declaration.

When you include the `+auto2protect` option, VirSim and third party tools are able to parse the encrypted source files without decoding the encrypted source description that follows these lists. For this reason Synopsys recommends the use of `+auto2protect` option instead of `+autoprotect` option.

When you include the `+auto2protect` option:

- VCS creates new files for the Verilog source files you specify on the `vcs` command line.

- VCS inserts the ``protected` compiler directive after the module header and the ``endprotected` compiler directive before the `endmodule` keyword. VCS encrypts the Verilog source between the ``protected` and ``endprotected` compiler directives that it inserts.

The following is an example of a Verilog source file whose entire contents you want to encrypt with source protection:

```
module top( inp, outp);
parameter size=8;
input [size-1:0] inp;
output [size-1:0] outp;
reg [7:0] count;
assign outp = count;
parameter stoptime=100;
always
begin:counter
    reg [size-1:0] int;
    count = 0;
    int = inp;
    while (int)
        begin
            if (int [0]) count = count + 1;
            int = int >> 1;
        end
        #stoptime $stop;
    end
endmodule
```

The following is an example of the contents of the new source file VCS creates when you run source protection with the `+auto2protect` option:

```
module top( inp, outp);
`protected
\,MGH?JR[D?0R_D#+XJ(MQD#HgXV@ZUVI2+HT)1OS)C8#L70A[9ge#^#5@WO0P_<
,Y)[^ZVDDCBf<EB?2(=)>S#aSR58?]Qgg6\#00f<^&#5.+RK[6<#2&X>SZM:)F9>
VLf:FHRsd[QP=WCC\gA;=g5M=>PG5EDUaZ:#/If[CTXV9RKJNNOf]>Cfgg[4&W.f
=2FD]<,R0?@:B0R:? \4fP_dgaGgF_IB9MV#E1M?b2)Cd._<:&@,KV\a5:Q3D]CPL
[9HDe2.gQYL0;\_Y^V\a;_Ag-fP;+K\;GUU/:HXFf;gaGJ1fO8_f1M)eGF8LRRY]
```

```

CB75+Q/]R_IAP/>HS+b<XFP,-BHfcZTIG0-QILLIa1#.RbX6.K?Oc8f5]f);BW=Q
FVa@-^&@e+:K0>(?(&ZL:4Z:.F[<5J]gQ+CaA]^7\.N^/0@RRZQe-]A,Z+L>?FQG
([A0S=LXHPgN[<Dg5fQG?^6IUP7.VV3a@$
`endprotected
endmodule

```

The following is an example of the contents of the new source file VCS creates when you run source protection with the `+auto3protect` option:

```

module top( inp, outp);
parameter size=8;
`protected
\,MGH?JR[D?0R_D#+XJ(MQD#HgXV@ZUVI2+HT)1OS)C8#L70A[9ge#^#5@W00P_<
,Y)[^ZVDDCBf<EB?2(=)>S#aSR58?]Qgg6\#00f<^&#5.+RK[6<#2&X>SZM:)F9>
VLf:FHRsd[QP=WCC\gA;=g5M=>PG5EDUaZ:#/If[CTXV9RKJNNOf]>Cfgg[4&W.f
=2FD]<,R0?@:B0R:?\4fP_dgaGgF_IB9MV#E1M?b2)Cd._<:&@,KV\a5:Q3D]CPL
[9HDe2.gQYL0;\_Y^V\a;_Ag-fP;+K\;GUU/:HXFf;gaGJ1fO8_f1M)eGF8LRRY]
CB75+Q/]R_IAP/>HS+b<XFP,-BHfcZTIG0-QILLIa1#.RbX6.K?Oc8f5]f);BW=Q
FVa@-^&@e+:K0>(?(&ZL:4Z:.F[<5J]gQ+CaA]^7\.N^/0@RRZQe-]A,Z+L>?FQG
([A0S=LXHPgN[<Dg5fQG?^6IUP7.VV3a@$
`endprotected
endmodule

```

Notice that parameter size is not encrypted but parameter stoptime is.

The following is an example of the contents of the new file VCS creates when you run source protection with the `+autoprotect` option:

```

module top
`protected
>Z.>B/)c?G.-UDVP<^?H0Yb_HSKg0>@UY0+-eY-/@YK^^Q3g+K6c\\5#ge+/8McF
WaK-/QI-?Gc^d8.))aeZ1gQBZbE_1.[U;1dVZ9b#. (A336;]&NM^GF,dR34)I]GZ
FEc-R?e1V3Mb1SUUB,J7R4[T\LZe?BDb#^@7/_5IAdM<(J.BN7\ [<[^We?@JeVSf
FKC)D.#bB@C#L^I4(55SG>:SXgTL^&GCG<&&6bQO;;EG)S]2d7X6,U:,<:S:<<KM
=&<OWaR?Ef.SEQW;RKYD.F\Se&WE+ALI=PafD8b:T[4#gbK??Igj@4c&0?#H]1:1
4HJ)e)G^U4:Y)^4@48:Gf:I6Ue)NG\JH:@fL5Q3?Gc+R)#)9f3g-cA(CM2fTf&(:
_+HVA=2@:MR&^=>5EbYF63a\BQ71<H>\GMJe&,PIA&I_9O4fMd7#PNW1?KgK2UP
XbVJ+FB\;6IY)+QZQTBS<+f;c:YeOc7eI(G:cT#1&7E]Jda1^)]H@cNWM$
`endprotected
endmodule

```

In this file encryption begins after the module identifier instead of after the port list.

Specifying Encrypted Filename Extensions

You can specify a different extension for the new filenames by adding the new extension as a suffix to the `+protect`, `+auto2protect`, `+autoprotect`, or `+sdfprotect` options.

For example, if you include the `+protect.protected` option on the `vcs` command line, for the source file named `my_design.v`, the encrypted new filename is `my_design.v.protected`.

Specifying Encrypted File Locations

You can specify the directory where VCS writes the new and encrypted files with the `+putprotect+` option followed by the path to the directory where you want VCS to create the new files. Include this option along with the `+protect`, `+auto2protect`, `+autoprotect`, or `+sdfprotect` option on the `vcs` command line.

Make sure there are no spaces between the `+putprotect+` option and the path to this directory. For example, with the following command line:

```
vcs cache.v +auto2protect.encrypt +putprotect+/u/design/cache
```

VCS creates the encrypted file `cache.v.encrypt` in the directory `/u/design/cache`.

Multiple Runs and Error Handling

When VCS creates encrypted files:

- If the encrypted file already exists, VCS outputs an error message and does not overwrite the existing file. This error message is:
`Error: file 'filename.vp' already exists.`
- If an error condition exists in the source file, VCS continues to create the encrypted file. This file might be corrupt.

To circumvent these problems you can use the `+deleteprotected` source protection option. This option disables the check for an existing encrypted file and deletes the encrypted file if an error condition occurred in that file.

Permitting CLI/PLI Access to Encrypted Modules

The `+pli_unprotected` option can be used to disable the normal PLI/CLI 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 option is used in conjunction with the `+protect`, `+autoprotect` or `+auto2protect` options. This option only affects the modules in the current file(s). Modules in files which were encrypted without using this flag are still fully protected.

This option only affects encrypted verilog files. It is ignored when encrypting SDF files.

Note: Turning off PLI/CLI protection will allow users of your modules to access object names and values, etc. In essence, the source code for your module could be substantially reconstructed using the CLI debugging commands and PLI `acc` routines.

Simulating Encrypted Models

When you simulate an encrypted model, information about hierarchical names and design objects is protected. This means that some CLI commands, such as those that display hierarchical names and design objects at a specified level of the hierarchy, do not work at protected levels of the design hierarchy. Protected levels are the encrypted levels.

Certain system tasks continue to work in protected levels of the design hierarchy but you need to know the hierarchical names in these protected regions and there is no way, other than from the vendor that encrypted the model, that you can obtain the hierarchical names in protected levels.

The `-Xman` compile-time option does not work in protected source descriptions.

Using the CLI

Some CLI commands whose arguments are design objects or instances do not work when these objects are in protected regions. These CLI commands are:

`info`

`scope`

`show`

Note:

The `info` command continues to display the simulation time.

Some CLI commands whose arguments are design objects continue to work but only if you know the hierarchical name of the object. These CLI commands are:

<code>always</code>	<code>break</code>	<code>force</code>	<code>once</code>
<code>print</code>	<code>release</code>	<code>set</code>	<code>tbreak</code>

You must know the hierarchical path names to use these commands.

Using System Tasks

The following system tasks continue to display design information from a protected region of a design:

`$display` `$write` `$monitor` `$strobe` `$gr_waves`

Note:

For `$display` the `%m` format specification does not work in protected regions

Writing PLI Applications

PLI access is restricted for encrypted modules. Any module which has any portion of it protected will cause the entire module to be deemed protected.

All `acc_next_...` type routines are blocked on protected modules. For example, `acc_next_port()` will immediately return NULL when operating on an instance whose module definition is protected.

The routine `acc_object_of_type()` can be used to determine if a module (or macromodule or primitive) is protected. Usage is:

```
if (acc_object_of_type(obj, accProtected)) {  
    printf("object %s is from a protected module\n",  
        acc_fetch_name(obj));  
}
```

If the object passed to `acc_object_of_type()` is a design object in a module, a port or register for example, the status of the parent module will be returned.

Mangling Source Files

The purpose of mangling is to create a Verilog source file that is functionally the same as an original Verilog file or set of Verilog files, but in the new Verilog source file the identifiers (module, instance, or signal names) no longer make any sense and all the comments are removed, making it very difficult to understand the design contained in the new file.

Mangling does not change the structure of the source description, in the new mangled file the keywords and syntax remain the same as those in the original Verilog file or set of Verilog files.

One use for this new Verilog source file is that you can send the mangled file to VCS_support@synopsys.com so that our Corporate Applications Engineers can examine problems you are experiencing while maintaining the only intelligible copy of your design at your site.

The name of the mangled source file you create is `tokens.v`.

You mangle your source description with the `-Xmangle` or `-Xmangle=number` option. When you enter this option, VCS creates a file named `tokens.v` that contains the mangled source description.

You can substitute `-Xman` for `-Xmangle`. The argument *number* can be 1 or 4:

`-Xman=1`

Randomly changes names and identifiers to provide a more secure code.

`-Xman=4`

Preserves variable names but removes comments.

`-Xman=12`

Does the same as `-Xman=4` but also adds comments about the system tasks in the source and the source file and line numbers of the module headers.

`-Xman=28`

Does the same as `-Xman=12` but adds a statistical analysis of the design.

The following is an example of Verilog source description before mangling:

```
module demo (modulus1,cpuData);
input [7:0] modulus1;
output [255:0] cpuData;
integer cpuTmpCnt;
reg [0:34] iPb[0:10]; //incoming pbus data buffer
reg [255:0] cpuDatareg;
assign cpuData = cpuDatareg;
function [0:255] merge_word;
input [0:31] source_word;
input [0:2] word_index;
begin
end
endfunction
initial begin
cpuDatareg = 256'h0;
for(cpuTmpCnt = 0; cpuTmpCnt<8; cpuTmpCnt=cpuTmpCnt+1)
begin : assemble_incoming
reg [0:34] inData35;
```

```

inData35=iPb[cpuTmpCnt];
$display("\tiPb[%0h]=%b, %h", cpuTmpCnt,
iPb[cpuTmpCnt],iPb[cpuTmpCnt] >> 3);

cpuDatareg=merge_word(cpuData,inData35[0:31],cpuTmpCn
t);
end
end
endmodule

```

The following is the mangled code in tokens.v produced by the `-Xmangle` or `-Xmangle=1` option:

```

`portcoerce
`inline
// No timescale specified
module Memoe(Remoe, Afmoe);

input [7:0] Remoe;
output [255:0] Afmoe;

integer Ifmoe;
reg [255:0] Wfmoe;

reg [0:34] Sfmoe[0:10];

assign Afmoe = Wfmoe;

function [0:255] Hgmoe;

input [0:255] Sgmoe;
input [0:31] Ehmoe;
input [0:2] Qhmoe;

reg [0:255] Sgmoe;
reg [0:31] Ehmoe;
reg [0:2] Qhmoe;
begin
end
endfunction

```

```

initial begin
Wfmoe = 256'b0;
for (Ifmoe = 0; (Ifmoe < 8); Ifmoe = (Ifmoe + 1)) begin : Bimoe

reg [0:34] Timoe;
Timoe = Sfmoe[Ifmoe];
$display("iPb[%0h]=%b, %h", Ifmoe, Sfmoe[Ifmoe],
(Sfmoe[Ifmoe] >> 3));
Wfmoe = Hgmoe(Afmoe, Timoe[0:31], Ifmoe);
end
end
endmodule

```

Notice that comments are not included in the mangled file.

The following is the mangled code in tokens.v produced by the -Xmangle=4 option:

```

`portcoerce
`inline
// No timescale specified
module demo(modulus1, cpuData);

    input[7:0]    modulus1;
    output[255:0] cpuData;

    integer      cpuTmpCnt;
    reg [255:0]  cpuDatareg;

    reg [0:34]   iPb[0:10];

    assign cpuData = cpuDatareg;

    function [0:255] merge_word;

    input[0:255] source_line;
    input[0:31]  source_word;
    input[0:2]   word_index;

```

```

    reg [0:255] source_line;
    reg [0:31] source_word;
    reg [0:2] word_index;
    begin
    end
endfunction

initial begin
    cpuDatareg = 256'b0;
    for (cpuTmpCnt = 0; (cpuTmpCnt < 8); cpuTmpCnt =
(cpuTmpCnt + 1))
        begin : assemble_incoming

            reg[0:34] inData35;
            inData35 = iPb[cpuTmpCnt];
            $display("iPb[%0h]=%b, %h", cpuTmpCnt,
iPb[cpuTmpCnt],
                (iPb[cpuTmpCnt] >> 3));
            cpuDatareg = merge_word(cpuData, inData35[0:31],
cpuTmpCnt);
        end
    end
endmodule

```

The comment is removed and default compiler directives and a comment about timescales is added.

The following is the mangled code in tokens.v produced by the `-Xmangle=12` option:

```

/* System Tasks:
    $display

*/

`portcoerce
`inline
/* Source file "mangle2.v", line 1 */
// No timescale specified
module demo(modulus1, cpuData);

```



```

input[7:0]    modulus1;
output[255:0] cpuData;

integer      cpuTmpCnt;
reg [255:0]  cpuDatareg;

reg [0:34]   iPb[0:10];

assign cpuData = cpuDatareg;

function [0:255] merge_word;

input[0:255] source_line;
input[0:31]  source_word;
input[0:2]   word_index;

reg [0:255]  source_line;
reg [0:31]   source_word;
reg [0:2]    word_index;
begin
end
endfunction

initial begin
    cpuDatareg = 256'b0;
    for (cpuTmpCnt = 0; (cpuTmpCnt < 8); cpuTmpCnt =
(cpuTmpCnt + 1))
        begin : assemble_incoming

            reg[0:34] inData35;
            inData35 = iPb[cpuTmpCnt];
            $display("iPb[%0h]=%b, %h", cpuTmpCnt,
iPb[cpuTmpCnt],
                (iPb[cpuTmpCnt] >> 3));
            cpuDatareg = merge_word(cpuData, inData35[0:31],
cpuTmpCnt);
        end
    end
endmodule

```

Here there are additional comments about the system tasks used and the source file and line number of the module header.

The following is the mangled code in tokens.v produced by the `-Xmangle=28` option:

```
/* System Tasks:
   $display

*/

`portcoerce
`inline
/* Source file "mangle2.v", line 1 */
// No timescale specified
module demo(modulus1, cpuData);

    input[7:0]    modulus1;
    output[255:0] cpuData;

    integer      cpuTmpCnt;
    reg [255:0]   cpuDatareg;

    reg [0:34]   iPb[0:10];

    assign cpuData = cpuDatareg;

    function [0:255] merge_word;

    input[0:255] source_line;
    input[0:31]  source_word;
    input[0:2]   word_index;

    reg [0:255]  source_line;
    reg [0:31]   source_word;
    reg [0:2]    word_index;
    begin
    end
endfunction

    initial begin
```

```

        cpuDatareg = 256'b0;
        for (cpuTmpCnt = 0; (cpuTmpCnt < 8); cpuTmpCnt =
(cpuTmpCnt + 1))
            begin : assemble_incoming

                reg[0:34] inData35;
                inData35 = iPb[cpuTmpCnt];
                $display("iPb[%0h]=%b, %h", cpuTmpCnt,
iPb[cpuTmpCnt],
                    (iPb[cpuTmpCnt] >> 3));
                cpuDatareg = merge_word(cpuData, inData35[0:31],
cpuTmpCnt);
            end
        end
    endmodule

```

```

// =====
//  DESIGN STATISTICS
//  =====
//
//                               Static<!!>  Elaborated<@>
//                               -----  -----
// No. of module instances:                1                1
// No. of comb. udp instances:              0                0
// No. of seq. udp instances:               0                0
// No. of gates:                           0                0
// No. of continuous assignments:          1                1
// No. of module+udp port connects:        0                0
// No. of registers (vector+scalar):       7                7
// No. of memories:                        1                1
// No. of scalar nets:                     0                0
// No. of vector nets:                     2                2
// No. of named events:                    0                0
// No. of always blocks:                   0                0
// No. of initial blocks:                  1                1
// No. of fork/join blocks:                 0                0
// No. of verilog tasks:                   0                0
// No. of verilog functions:                1                1
// No. of verilog task calls:              0                0
// No. of verilog function calls:          1                1
// No. of system task calls:               1                1
// No. of user task calls:                 0                0
// No. of system function calls:           0                0

```

```

// No. of user function calls:           0           0
// No. of hierarchical references:       0           0
// No. of concatenations:               0           0
// No. of force/release statements:      0           0
// No. of bit selects:                   0           0
// No. of part selects:                   1           1
// No. of non-blocking assignments:      0           0
// No. of specify blocks<#>:            0           0
// No. of timing checks:                 0           0
//
// No. of top level modules:             1
//      modules:                         1
// No. of udps:                           0
//      seq. udps:                       0
//      comb. udps:                      0
// No. of module+udp ports:              2
// No. of system tasks:                  1
// No. of user tasks:                    0
// No. of system functions:              0
// No. of user functions:                0
//
// Footnotes:
// -----
// <!> No. of unique instances of a construct as it appears
// in the source.
// <@> No. of instances of a construct when the design //
// is elaborated.
// <#> Multiple specify blocks in the SAME module are //
// combined and counted as ONE block.

```

Note: If you are creating a mangled tokens.v file to send to Synopsys for technical support, we advise checking to see if you can duplicate the problem using the tokens.v file. Also please include with this file a copy of the vcs and simv command lines that duplicate the problem and any SDF files that need to be back annotated and C source files or object files for PLI applications that need to be linked to the simv executable.

Creating A Test Case

Here is a quick way to create a small test case:

1. Remove `-o` option if you are using it.
2. Add the `-Xman=4` option to your vcs command line (in addition to whatever other options you are using) and run VCS.
VCS will not actually compile your design, instead it generates a file called `tokens.v`.
3. Rename the `tokens.v` file to `tokens.orig`
4. Run the following command line:

```
vcs tokens.orig
```

5. Cut out the problem module from `tokens.orig` and paste it into a new file, say `test.v`.
6. Run the following command line:

```
vcs -Xman=4 test.v -v tokens.orig
```

This will create a new `tokens.v` file

7. Run the following command line again:

```
vcs tokens.v
```

Do this to verify that the problem still exists.

8. Zip the `tokens.v` file
9. FTP the `tokens.v.gz` file to us:
 - a. Connect to our FTP site by entering:

```
ftp ftp.synopsys.com
```

b. At the Name prompt, enter anonymous.
At the Password prompt, enter your e-mail address.

c. At the `ftp>` prompt enter:

```
cd incoming
```

d. Enter the following FTP command:

```
binary
```

e. Enter the following FTP command:

```
put tokens.v.gz
```

10. Now send e-mail to VCS_support@synopsys.com informing us that you have uploaded the test case.

Preventing Mangling of Top-Level Modules

You can prevent the mangling of top-level modules in your source description with the `-Xnomangle` option. The syntax for this option is:

```
-Xnomangle=.first | module_identifier  
[, module_identifier...]
```

For example:

```
-Xnomangle=.first
```

Prevents the mangling of the first module VCS encounters in the source description.

```
-Xnomangle=top1, top2
```

Prevents the mangling of modules `top1` and `top2`

The `-Xnomangle` option should only be used on top level modules, i.e., ones that are not called by any other module in the design being mangled. The code which performs the mangling knows how to identify module and port declarations, but not module references. Thus, if you exempt a module which is referenced by some other module, the reference will try to use the mangled name instead of the original (and fail).

Source Protection

25-26

A

VCS Environment Variables

There are a number of variables you use to set up the simulation environment in VCS.

This appendix covers the following topics:

- [Simulation Environment Variables](#)
- [Optional Environment Variables](#)

Simulation Environment Variables

ERROR_WHEN_UNBOUND 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
```

SYSTEMC

Specifies the location of the SystemC simulator used with the VCS / SystemC cosimulation interface. See [Chapter 19, "Using the VCS / SystemC Cosimulation Interface"](#).

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:

```
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 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 30 days before a license expires. You can specify that this warning 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_RUNTIME

Specifies which runtime library named libvcs.a VCS uses. This variable is for solving problems that require patches from VCS and should not be set by default.

VCS_SWIFT_NOTES

Enables the printf PCL command. PCL is the Processor Control Language that works with SWIFT microprocessor models. To enable it, set this environment variable's value to 1.

B

Compile-Time Options

The `vcs` command performs compilation of your designs and creates a simulation executable. Compiled event code is generated and used by default. The generated simulation executable, called `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 Verilog source files.

The following is an example command line used at compile time:

```
% vcs top.v toil.v +v2k
```

For complete usage information on the `vcs` command, see [Chapter 3, "Compiling Your Design"](#).

This appendix lists the following:

- [Options for Accessing Verilog Libraries](#)
- [Options for Incremental Compilation](#)
- [Options for Help and Documentation](#)
- [Options for SystemVerilog](#)
- [Options for OpenVera 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 Debugging
- Options for Finding Race Conditions
- Options for Starting Simulation Right After Compilation
- Options for Compiling OpenVera Assertions (OVA)
- Options for Compiling For Simulation With Vera
- Options for Compiling For Coverage Metrics
- Options for Discovery Visual Environment and UCLI
- Options for Converting VCD and VPD Files
- Options for Specifying Delays
- Options for Compiling an SDF File
- Options for Profiling Your Design
- Options for File Containing Source File Names and Options
- Options for Compiling Runtime Options into the simv Executable
- Options for Pulse Filtering
- Options for PLI Applications
- Options to Enable and Disable Specify Blocks and Timing Checks
- Options to Enable the VCS DirectC Interface
- Options for Negative Timing Checks
- 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 Assembler](#)
- [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” on page B-57](#)
- [Options to Specify the Time Scale](#)
- [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.

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, 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.

`+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. You enable creating these temporary object files, and specify how many module definitions are in these files, with this option. 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.

`-Mupdate [=0]`

By default, VCS overwrites the makefile between compilations. If you wish to preserve the makefile between compilations, enter the 0 argument with this option. Entering this option without the 0

argument specifies the default condition, which is incremental compilation and updating the makefile.

`-Mdefine=name=value`

Adds a definition to the makefile.

`-Mdelete`

Use this option for the rare occurrence when the `chmod -x simv` command in the makefile can't change the permissions on an old `simv` executable. This option replaces this command with the `rm -f simv` command in the makefile.

`-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`.

`-Mfilename=prefix`

Base name (prefix) for C source and object files.

`-Minclude=filename`

Adds an include statement to the makefile.

`-Mldcmd=value`

Format string used to invoke the linker directly.

`-Mlib=dir`

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. It 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 a design (see [“Incremental Compilation” on page 3-3](#)). You can specify more than one place for VCS to look for descriptor information and object files by providing multiple arguments with this option. For 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, for example:

```
vcs design.v -Mlib=/design/dir1:/design/dir2
```

`-Mloadlist=value`

Directly invokes the linker to link programs if *value* is Yes.

`-Mmakefile=filename`

Names the generated makefile (default is makefile).

`-Mmakeprogram=program`

Specifies the make program that analyzes dependencies, compiles what needs compiling, links, builds, and performs the necessary steps to produce the executable file that you run to simulate your design with VCS.

Commonly used make programs include the make program that comes with your operating system and gmake from the GNU foundation. The make program from your operating system is the default make program.

If you are including options for the make program then include the entire argument in quotation marks. Also you can shorten this option to `-Mmakep`.

The following example shows how to include the `-j` option to the gmake program to specify parallel compilation:

```
-Mmakep="gmake -j 4"
```

In this example the 4 argument to the `-j` option for gmake specifies using four CPUs for parallel compilation. You can also use the VCS `-j number_of_CPUs` compile-time option to do this.

`-Mrelative_path`

Use this option if your linker has a limitation on the length of the linker line in the make file. If you specify a relative path with the

`-Mlib` option, the `-Mrelative_path` option does not expand the relative path to an absolute path on the linker line in the make file.

`-Msrclist=filename`

Specifies the name of source list file that lists all object files created by VCS. The default is `filelist`.

`-noIncrComp`

Disable incremental compilation.

Options for Help and Documentation

`-h` or `-help`

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

`-doc`

Starts a PDF file reader to display the PDF file for the VCS Documentation Navigator. This option tells VCS to execute the following system command:

```
"$PDF_READER $VCS_HOME/doc/UserGuide/navigator.pdf"
```

By default, `PDF_READER` is set to the first `acroread` executable (the Adobe Acrobat reader) that it finds in the directories specified for your `PATH` environment variable. You can set the `PDF_READER` environment variable to other PDF file display tools, for example `xpdf` on Linux.

Options for SystemVerilog

`-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:

`disable`

Disables all SystemVerilog assertions in the design.

`enable_diag`

Enables further control of results reporting on SystemVerilog assertions with runtime options.

`filter_past`

Ignores assertions defined with the `$past` system task when the past history buffer is empty. For instance, at the very beginning of the simulation, the past history buffer is empty. So 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_cover`

Disables coverage for cover statements.

`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.

`-sv_pragma`

Tells VCS to compile the SystemVerilog Assertions code that follows the `sv_pragma` keyword in a single line or multi-line comment.

Options for OpenVera Native Testbench

`+debug_all`

Enables you to use the OpenVera testbench GUI.

`-ntb`

Enables the use of the OpenVera testbench language constructs described in the *OpenVera Language Reference Manual: Native Testbench*.

`-ntb_cmp`

Compiles and generates the testbench shell (*file.vshell*) and shared object files. Use this option when compiling the `.vr` file separately from the design files.

`-ntb_define macro`

Specifies any OpenVera macro name on the command line. You can specify multiple macro names using the plus (+) character.

`-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_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`

Reports errors, during compilation or simulation, when there is an out-of-bound or illegal array access.

`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 or in the specified file. Enter this argument with the `dep_check` argument.

`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 result 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]`.

For example, if “adder” is the name of the interface, specifying `-ntb_opts vera_portname` causes VCS to use the following shell module:

```
vera_shell vshell(  
    .SystemClock (SystemClock),  
    .adder_inp1 (inp1),  
    .adder_inp2 (inp2),
```

```
    ...  
);
```

Without `vera_portname`, the Vera shell module name is based on the name of the OpenVera program by default. Bind signals are named, for example, as: `\ifc.signal[3:0]`. The interface ports are named `\ifc.signal`.

In this example, assume that “adder” is the name of the OpenVera program:

```
adder_test vshell(  
    .SystemClock (SystemClock),  
    .\adder.inp1      (inp1),  
    .\adder.inp2      (inp2),  
    ...  
);
```

You can enter more than one keyword argument, using the plus (+) character. For example:

```
-ntb_opts use_sigprop+vera_portname
```

```
-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`, `.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.

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 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 this option, you can omit the `+v2k` option.

`+verilog1995ext+ext`

Specifies a file name extension for Verilog 1995 files. Using this option allows you 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. See [“Initializing Memories and Regs” on page 3-8](#).

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

Initializes all bits of all regs in the design. See [“Initializing Memories and Regs” on page 3-8](#).

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 [“Applying Radiant Technology to Parts of the Design” on page 3-36](#).

Options for 64-bit Compilation

`-full64`

Enables compilation and simulation in 64-bit mode. See [“Initializing Memories and Regs” on page 3-8](#).

`-comp64`

Enable 64 bit compilation that generates a 32-bit simv executable. See [“Initializing Memories and Regs” on page 3-8](#).

Options for Debugging

`-line`

Enables source-level debugging tasks such as stepping through the code, displaying the order in which VCS executed lines in your code, and the last statement executed before simulation stopped. Typically you enter this option with a `+cli` option.

For example: `vcs +cli+1 -line`

`+cli+[module_identifier=]level_number`

Enables command line interface (CLI) interactive debugging commands. Using this option creates the Direct Access Interface Directory, by default named `simv.daidir`, in the directory where VCS creates the executable file. See [“The Direct Access Interface Directory” on page 3-7](#).

The level number enables more and more commands. The level number can be any number between 1 and 4:

`+cli+1` or `+cli`

Enables commands that display the value of nets and registers and deposit values on registers

`+cli+2`

Same as above, plus enables callbacks on signals, that is commands that do something when there is a value change.

For example:

```
break signal_name
```

`+cli+3`

Same as above, plus enables force and release of nets. For example:

```
force net=0
```

This level number does not enable forcing values on registers.

+cli+4

Same as above, plus enables force and release of registers.

Using these options also creates the Direct Access Interface Directory, by default named `simv.daidir`, in the directory where VCS creates the executable file.

You can include a module identifier (name) in this option to enable a level of CLI commands for all instances of the module. If you want to enable CLI commands in other modules, enter another +cli option, for example:

```
vcs source.v +cli+dev1=4 +cli+dev2=4
```

This command line enables level four CLI commands in all instances of modules `dev1` and `dev2`.

+cliedit

In UNIX, enables tcsh-like CLI mode. This mode enables you to use the GNU command line editing interface for entering CLI commands. For example, with this mode, entering Control - p displays the previous CLI command at the CLI prompt.

To use this option you first must download a zipped tar file from the Synopsys FTP site, unzip it, extract this tar file, and set the `GNURL_HOME` environment variable. You may want to install this file at a location accessible to all users at your site. To do so:

1. Enter on a command line `FTP FTP.synopsys.com`
2. At the FTP Name prompt enter `anonymous.`
This enables a guest login.
3. Enter your E-mail address as your password.
4. Enter the FTP command `cd TOOLS` to move to the directory that contains the tar file.

5. Enter the FTP command `bin`.
6. Enter the FTP command `get readline-2.0.tar.gz`. This downloads the file to your current directory.
7. Enter the FTP command `quit` to exit FTP.
8. Unzip the tar file by entering:


```
gunzip readline-2.0.tar.gz.
```
9. Extract the tar file by entering, for example:


```
tar xvf readline-2.0.tar
```
10. This creates the `readline-2.0` directory. In that directory is the `make_gnurl` script.
11. Execute the `make_gnurl` script. This script builds the `gnurl.o` and other object files that you need for this option. It creates a subdirectory that it names after the UNIX platform you use, for example, `sparc`. It writes the `gnurl.o` and other object files in that subdirectory.
12. Set the `GNURL_HOME` environment variable to the subdirectory created by the script, for example:


```
setenv GNURL_HOME /u/eng/readline-2.0/sparc
```

 Instead of setting this environment variable you can copy the `gnurl.o` and other object files in the subdirectory to the `lib` directory in your VCS installation.

Documentation for this mode is included in the `readline-2.0/doc` directory.

+acc+level_number

Old method to enable PLI ACC capabilities for the entire design. Synopsys recommends that you do not use this option. 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, such as applying breakpoints to signals or stepping through the code in certain parts of your design, during a previous simulation of the design. ACC capabilities enable debugging operations but slow down your simulation so you only want to apply them where you need them.

You record where you last used them, in a previous simulation, with the `+vcs+learn+pli` runtime option. This option records where you used ACC capabilities in a file named `pli_learn.tab`. If you do not change the file's name or location, you can omit `+filename` from the `+applylearn` compile-time option.

Options for Finding Race Conditions

`+race`

Specifies that VCS generate a report of all the race conditions in the design and write this report in the `race.out` file during simulation. See [“The Dynamic Race Detection Tool” on page 11-2](#).

`+race=all`

Analyzes the source code during compilation to look for coding styles that cause race conditions. See [“The Static Race Detection Tool” on page 11-13](#).

`+racecd`

Specifies that during simulation VCS generate a report of the race conditions in the design between the ``race` and ``endrace` compiler directives and write this report in the `race.out` file. See [“The Dynamic Race Detection Tool” on page 11-2](#).

`+race_maxvecsize=size`

Specifies the largest vector signal for which the dynamic race detection tool looks for race conditions.

Options for Starting Simulation Right After Compilation

`-R`

Runs the executable file immediately after VCS links it together.

`-s`

Specifies stopping simulation just as it begins and entering the CLI interactive mode. Use this option on the `vcs` command line along with the `-R` and `+cli` options. The `-s` option is also a runtime option on the `simv` command line.

`-i filename`

Specifies a file containing CLI commands that VCS executes when simulation starts. After the end of that file is reached, input commands are taken from the standard input. This option works only when you also include the `-R` and `-s` options. It is normally entered with the `+cli+number` compile-time option. This option is also accepted by the output `simv` executable; it is really a runtime option but it is frequently entered on the `vcs` command line.

Options for Compiling OpenVera Assertions (OVA)

`-ovac`

Starts the OVA compiler for checking the syntax of OVA files that you specify on the vcs command line. This option is for when you first start writing OVA files and need to make sure that they can compile correctly.

`-ova_cov`

Enables viewing results with functional coverage.

`-ova_cov_events`

Enables coverage reporting of expressions.

`-ova_cov_hier filename`

Limits functional coverage to the module instances specified in *filename*.

`-ova_debug | -ova_debug_vpd`

Enables viewing results with DVE.

`-ova_file filename`

Identifies *filename* as an assertion file. It is not required if the file name ends with `.ova`. For multiple assertion files, repeat this option with each file.

`-ova_filter_past`

Ignores assertions defined with the past operator when the past history buffer is empty. For instance, at the very beginning of the simulation the past history buffer is empty. So, a check/forbid at the first sampling point and subsequent sampling points should be ignored until the past buffer has been filled with respect to the sampling point.

`-ova_enable_diag`

Enables further control of result reporting with runtime options.

- ova_inline
Enables compiling of OVA code that is written inline with a Verilog design.
- ova_lint
Enables general rules for the OVA linter.
- ova_lint_magellan
Enables Magellan rules for the OVA linter.

Options for Compiling For Simulation With Vera

- vera
Specifies the standard VERA PLI table file and object library.
- vera_dbind
Specifies the VERA PLI table file and object library for dynamic binding.

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 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.

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 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 `^` bitwise XOR and `~^` 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. 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 and a `cmView` command line option.

`-cm_fsmcfg filename`

Specifies an FSM coverage configuration file.

`-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.

`-cm_hier filename`

When compiling for line, condition, toggle or FSM coverage, 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.

`-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 filename`

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_opfile filename`

Specifies a file that contains a list of signals that you want to also treat as observation points.

`-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.

`batch`

Tells VCS to start cmView to write reports in batch mode. This is the default operation.

You enter cmView command line options to the right of this option and its argument.

`-cm_resetfilter filename`

Filters out FSM coverage 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.

`-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-  
testshell.v"
```

`-cm_tglfile filename`

Specifies displaying at runtime a total toggle count for one or more subhierarchies specified by the top-level module instance entered in the file. This option is also a runtime option.

`-cm_tgl mda`
Enables toggle coverage for Verilog-2001 multidimensional arrays (MDAs) and SystemVerilog unpacked MDAs. Not required for SystemVerilog packed MDAs.

Options for Discovery Visual Environment and UCLI

`-debug`
Enables DVE and UCLI debugging. This option does not enable line stepping.

`-debug_all`
Enables DVE and UCLI debugging including line stepping.

`-ucli`
Forces runtime to go into UCLI mode by default.

`-gui`
When used at compile time, starts DVE at runtime.

`-debug_pp`
Enables minimal debug access so as to allow VPD dumping and assertion debug. You can view the results inside DVE in postprocessing mode.

Creates a VPD file (when used with the Verilog system task `$vcdpluson`) and enables DVE for postprocessing a design. Using `-debug_pp` can save compilation time by eliminating the overhead of compiling with `-debug` and `-debug_all`.

`-PP`
Enables you to enter system tasks like `$vcdpluson` in your Verilog source code to create a VPD file during simulation. This option minimizes net data details for faster post-processing.

`-assert dve`

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

Note:

The `-debug_pp` option writes a smaller VPD file than `-debug -PP` making for less overhead.

`+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.

`+vcdfile+filename`

Specifies the VCD file you want to use for post-processing.

`+vpdfileswitchsize+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

Options for Converting VCD and VPD Files

`-vcd2vpd vcd_filename vcdplus_filename`

Tells VCS to find and run the `vcd2vpd` utility that converts a VCD file to a VPD file. VCS inputs to the utility the specified VCD file and the utility outputs the specified VPD file.

`-vpd2vcd vcdplus_filename vcd_filename`

Tells VCS to find and run the `vpd2vcd` utility that converts a VPD file to a VCD file. VCS inputs to the utility the specified VPD file and the utility outputs the specified VCD file.

Options for Specifying Delays

`+allmtm`

Specifies enabling the `simv` executable for the `+mindelays`, `+typdelays`, or `+maxdelays` options at runtime, instead of at compile-time, to specify which of the minimum, typical, or maximum delay values to use during simulation from `min:typ:max` delay value triplets in module path delays and timing delays. This option is also used for compiling SDF files.

This option does not work for `min:typ:max` delay value triplets in other delay specification in your source code. Do not use this option with the `+maxdelays`, `+mindelays`, or `+typdelays` compile-time options. For more information, see [“Specifying Min:Typ:Max Delays at Runtime”](#) on page 13-38.

`+charge_decay`

Enables charge decay in `triereg` nets. Charge decay will not work if you connect the `triereg` 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 in delay specifications and also in delay entries in SDF files. See [“Min:Typ:Max Delays” on page 13-37](#).

`+mindelays`

Specifies using the minimum timing delays in min:typ:max delay triplets in delay specifications and also in delay entries in SDF files. See [“Min:Typ:Max Delays” on page 13-37](#).

`+typdelays`

Specifies using the typical timing delays in min:typ:max delay triplets in delay specifications and also in delay entries in SDF files. See [“Min:Typ:Max Delays” on page 13-37](#).

`+multisource_int_delays`

Enables the multisource INTERCONNECT feature, including transport delays with full pulse control. See [“INTERCONNECT Delays” on page 13-32](#).

`+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.

`+old_iopath`

By default VCS replaces negative module path delays in SDF files with a 0 delay. If you include this option, VCS replaces these negative delays with the delay specified in a module's `specify` block.

`-negdelay`

Enables the use of negative values in IOPATH and INTERCONNECT entries in SDF files.

`+transport_int_delays`

Enables transport delays for delays on nets with a delay backannotated from an INTERCONNECT entry in an SDF file. The default is inertial delays. See [“Transport and Inertial Delays” on page 12-2](#), [“Enabling Transport Delays” on page 12-7](#) and [“INTERCONNECT Delays” on page 13-32](#).

`+transport_path_delays`

Enables transport delays for module path delays. See [“Transport and Inertial Delays” on page 12-2](#) and [“Enabling Transport Delays” on page 12-7](#).

Options for Compiling an 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. See [“Specifying Min:Typ:Max Delays at Runtime” on page 13-38](#).

`+csdf+precompile`

Precompiles your SDF file into a format that VCS can parse when it compiles your Verilog code. See [“Precompiling an SDF File” on page 13-7](#).

`+csdf+precomp+dir+directory`

Specifies the directory path where you want VCS to write the precompiled SDF file. See [“Specifying an Alternative Name and Location” on page 13-8](#).

`+csdf+precomp+ext+ext`

Specifies an alternative to the “_c” character string addition to the file name extension of the precompiled SDF file. See [“Specifying an Alternative Name and Location” on page 13-8](#).

`+maxdelays`

Specifies using the maximum timing delays in min:typ:max delay triplets when compiling the SDF file. See [“Min:Typ:Max Delays” on page 13-37](#). The `mtm_spec` argument to the `$sdf_annotate` system task overrides this option. See [“The \\$sdf_annotate System Task” on page 13-3](#).

`+mindelays`

Specifies using the minimum timing delays in min:typ:max delay

triplets when compiling the SDF file. See [“Min:Typ:Max Delays” on page 13-37](#). The *mtm_spec* argument to the `$sdf_annotate` system task overrides this option. See [“The \\$sdf_annotate System Task” on page 13-3](#).

`-negdelay`

Enables the use of negative values in IOPATH and INTERCONNECT entries in SDF files.

`+oldsdf`

Disables compiling the SDF file. Use this option if you cannot meet the conditions for compiling SDF files. See [“Limitations on Compiling the SDF File” on page 13-5](#).

`+sdf_nocheck_celltype`

For a module instance to which an SDF file backannotates delay data, disables comparing the module identifier in the source code with the CELLTYPE entry in the SDF file. See [“Disabling CELLTYPE Checking in SDF Files” on page 13-15](#).

`+typdelays`

Specifies using the typical timing delays in min:typ:max delay triplets when compiling the SDF file. See [“Min:Typ:Max Delays” on page 13-37](#). The *mtm_spec* argument to the `$sdf_annotate` system task overrides this option. See [“The \\$sdf_annotate System Task” on page 13-3](#).

`+vcs+mipdexpand`

This option is used when backannotating SDF delay values from an ASCII text SDF file at runtime, as specified by the `+oldsdf` compile-time option, which disables compiling the SDF file during compilation. If the SDF file contains PORT entries for the individual bits of a port, using this option enables VCS to backannotate these PORT entry delay values. Similarly, using this compile-time option enables a PLI application to pass delay values to individual bits of a port.

As an alternative to using this option, you can use the `\vcs_mipdexpand` compiler directive or you can enter the `mipb` ACC capability in your PLI table file. For example:

```
$sdf_annotate call=sdf_annotate_call  
acc+=rw,mipb:top_level_mod+
```

When you compile the SDF file (recommended) the backannotation of the delay values for individual bits of a port does not require this option.

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. See [“Profiling the Simulation” on page 4-13](#).

`+vissymbols`

Makes symbols visible when you are using the `prof` or `gprof` program (not the VCS profiler that is enabled by the `+prof` option) to profile generated code.

Options for File Containing Source File Names and Options

`-f filename`

Specifies a file name that contains a list of absolute path names for Verilog source files and compile-time options.

You can enter in this file all the compile-time options that begin with a plus (+) character with three exceptions: do not enter the `+comp64`, `+full64`, or `+memopt` compile-time options.

You can also enter the following compile-time options that begin with a minus (-) character:

```
-f          -gen_asm      -gen_obj
-line      -l          -u          -v          -y
```

You can also enter the runtime options in this file so that VCS compiles them into the `simv` executable.

You cannot include C source or object file names, for PLI applications, in this file.

You can specify a path name for the `filename` argument. Note that you can also enter the `-f` option in this file with the path name of another file that also contains a list of absolute path names for Verilog source files and compile-time options.

`-F filename`

Similar to the `-f` option, but if you specify a path name for the `filename` argument, you do not have to list absolute path names for the Verilog source files that you list in the file. VCS uses the path to this file as the path to the Verilog source files.

Nested files are not supported.

`-file filename`

This option is for problems you might encounter with entries in files specified with the `-f` or `-F` options. This file can contain more compile-time options and different kinds of files. It can contain options for controlling compilation, PLI options and object files.

You can also use escape characters and meta-characters like `$`, `\`, and `!` in this file and they will expand. For example:

```
-CFLAGS '-I$VCS_HOME/include'  
/my/pli/code/$PROJECT_VERSION/treewalker.o  
-P /my/pli/code/$PROJECT_VERSION/treewalker.tab
```

You can comment out entries in this file with the Verilog `//` and `*` `*/` comment characters.

Options for Compiling Runtime Options into the simv Executable

`+plusarg_save`

Some runtime options must be preceded by the `+plusarg_save` option for VCS to compile them into the executable. You can specify this option either on the `vcs` command line or in the file specified with the `-f` or `-F` option.

`+plusarg_ignore`

Tells VCS not to compile the following runtime options into the `simv` executable. This option is typically used in the file that you specify with the `-f` compile-time option and is used to counter the `+plusarg_save` option on a previous line.

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. See [“Pulse Control” on page 12-7](#).

`+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. See [“Specifying the Delay Mode” on page 12-20](#).

Options for PLI Applications

`+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` options 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 and Disable Specify Blocks and Timing Checks

`+pathpulse`

Enables the search for the `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. See [“Enabling Negative Timing Checks” on page 14-13](#).

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 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 Negative Timing Checks

`+neg_tchk`

Enables negative values in timing checks. See [“Enabling Negative Timing Checks” on page 14-13](#).

`+old_ntc`

Prevents the other timing checks from using delayed versions of

the signals in the `$setuphold` and `$recrem` timing checks. See [“Other Timing Checks Using the Delayed Signals”](#) on page 14-14.

+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. See [“Checking Conditions”](#) on page 14-18.

+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. See [“Using Multiple Non-Overlapping Violation Windows”](#) on page 14-23.

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 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 `$dumpports` system task or the simulation history file for LSI certification specified by the `$lsi_dumpports` 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. See [Chapter 16, "SWIFT VMC Models and SmartModels"](#).

`-lmc-swift-template`

Generates a Verilog template for a SWIFT Model. See ["Generating Verilog Templates" on page 16-4](#).

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. See [“Using Lint” on page 3-10](#).

```
-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:

<code>+warn=noIPDW</code>	Enables all warning messages except the warning with the IPDW ID
<code>+warn=none,TFIPC</code>	Disables all warning messages except the warning with the TFIPC ID.
<code>+warn=noIPDW,noTFIPC</code>	Disables the warning messages with the IPDW and TFIPC IDs.
<code>+warn=all</code>	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` compile-time options. This option also enables full PLI access to these modules. This option also overrides a `nocelldefinepli` entry in the `.tab` files in the `vcs_install_dir/virsimdir` subdirectory.

`+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`

Disables 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` compile-time 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.

As an alternative to using the `+nocelldefinepli+1` option, you can add an entry in the `virsim.tab` AND `virsim_pp.tab` files (located in `$VCS_HOME/virsim_support`) to turn off VPD dumping for modules defined under the ``celldefine` compiler directive. Enter the keyword `nocelldefinepli` with appropriate spaces in the `$virsim` line. For example:

```
$virsim    data=0    check=vp_check_virsim
misc=vp_misc_virsim callback nocelldefinepli
acc+=rw,cbka:*
```

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 work station's platform, the

host ID of your work station (used in licensing), the version of the VCS compiler (same as VCS) and the VCS build date.

Options for Controlling the Assembler

`-gen_asm`

Enables assembly code generation mode. Use this option if you encounter problems in native code generation. Not supported on IBM RS/6000 AIX.

`-as assembler`

Selects an alternate assembler. Not supported on IBM RS/6000 AIX.

`-ASFLAGS options`

Passes options to assembler. Not supported on IBM RS/6000 AIX.

`-C`

Stops after generating the assembly code intermediate files. If you also enter the `-gen_asm` option, it does not assemble the assembly code files. Use this option if you want to assemble by hand. This option can also be used with the `-gen_c` options for disabling C code compilation.

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).
- `-syslib libs`
Specifies system libraries. For example `-syslib -ly -lc` includes the machine-specific system libraries. Normally, *libs* entered on the `vcs` command line using `-lname` option are placed in front of `libvcs.a` on the link line. The `-syslib` option tells VCS to place the specified *libs* after `libvcs.a` on the link line.

Options for Controlling the C Compiler

- `-gen_c`
Generates C language code. This is the default in IBM RS/6000 AIX.
- `-cc compiler`
Specifies an alternate C compiler
- `-CC options`
Passes options to the C compiler or assembler.
- `-CFLAGS options`
Pass 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` compile-time option. For example:

```
vcs source.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 HP, 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 if you also enter the `-gen_c` option, does not compile the C code files (`-gen_c` is the default on IBM RS/6000 AIX). Use this option if you want to compile by hand. This option can also be used with the `-gen_asm` option for disabling assembly.

`-O0`

Suppresses optimization for faster compilation (but slower simulation). Suppresses optimization both for how VCS writes intermediate C code files and how VCS 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. The `-O0` variant has special mention above.

`-override-cflags`

Tells VCS not to pass its default options to the C compiler. VCS has a number of C compiler options that it passes to the C compiler by default. The options it passes depends on the platform, whether it's a 64 or 64-32 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 CLI access to the modules in the protected source file being created (PLI and CLI 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 `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. See [“Changing Parameter Values From the Command Line”](#) on page 3-12.

`-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. See [“Changing Parameter Values From the Command Line” on page 3-12.](#)

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`

It may happen that 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, like 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.

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.

See "[Using the VCS / SystemC Cosimulation Interface](#)".

Reduce Memory Consumption

`+memopt [+2]`

Applies optimizations to reduce memory. See "[Initializing Memories and Regs](#)" on page 3-8.

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.

Treat Output Ports As Inout Ports

`+spl_read`

Tells VCS to treat output ports as “inout” in order to facilitate more accurate multi-driver contention analysis across module boundaries. This option can have an adverse impact on runtime performance.

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.

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 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.

`-cm_tgl mda`

Enables toggle coverage for Verilog-2001 MDAs and SystemVerilog unpacked MDAs. Not required for SystemVerilog packed MDAs.

Specifying a Log File

`-l filename`

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

Hardware Modeling

`-lmc-hm`

Compiles a design that instantiates a hardware model. Including this option is an alternative to specifying the `lmc.tab` PLI table file and the `lmc.o lmc_sfi.a` object file and library that you need for hardware modeling.

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 using this option:

```
set path=($VCS_HOME/bin\  
$VCS_HOME/`$VCS_HOME/bin/vcs -platform`/bin\  
$path)
```

Specifying Native Code Generation

-gen_obj

Generates object code; default on Solaris, HP and Linux platforms. Not supported on IBM RS/6000 AIX.

For Long Calls

-B

Generates long calls for large designs. Applies to HP 9000/700 models only.

Compile-Time Options

C-64

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. See [“Compiling Runtime Options Into the `simv` Executable” on page 3-21](#).

For complete usage information on the `simv` executable, see [Chapter 4, "Simulating Your Design"](#).

This appendix describes the following runtime options:

- [Options for Simulating OpenVera Testbenches](#)
- [Options for Simulating OpenVera Assertions](#)
- [Options for SystemVerilog Assertions](#)
- [Options for a CLI Command File](#)

- [Options for Specifying VERA Object Files](#)
- [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 Controlling \\$gr_waves System Task Operations](#)
- [Options for VCD Files](#)
- [Options for Specifying Min:Typ:Max Delays](#)
- [Options for Flushing Certain Output Text File Buffers](#)
- [Options for Licensing](#)
- [General Options](#)

Options for Simulating OpenVera Testbenches

`+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 value is less than 0. The value can be:

- 0 - continue
- 1 - exit on first error (default value)
- N* - exit on *n*th 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_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 CLI 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 Simulating OpenVera Assertions

`-ova_quiet [1]`

Disables printing results on screen. The report file is not affected. With the 1 argument, only a summary is printed on screen.

`-ova_report [filename]`

Generates a report file in addition to printing results on screen. Specifying the full path name of the report file overrides the default report name and location.

`-ova_verbose`

Adds more information to the end of the report, including assertions that never triggered, attempts that did not finish, and a summary with the number of assertions present, attempted, and failed.

`-ova_name name | pathname`

Specifies an alternative name, or location and name, for the

./simv.vdb/scov/results.db and ./simv.vdb/reports/ova.report files. Use this option if you want data and reports from a series of simulation runs. It is a way of preventing VCS from overwriting these files from a previous simulation. If you just specify a name, VCS creates the alternatively named files in the default directories. You can also specify a different location and name for these files by specifying the path using the slash character /. For example:

```
-ova_name /net/design1/ova/run2
```

This example tells VCS to write run2.db and run2.report in the /net/design1/ova directory.

The following runtime options control how VCS writes its report on OpenVera Assertions. You can use them only if you compiled with the `-ova_enable_diag` compile-time option.

```
-ova_filter
```

Blocks reporting trivial if-then successes. These happen when an if-then construct registers a success only because the `if` portion is false (and so the `then` portion is not checked). With this option, reporting only shows successes in which the whole expression matches.

```
-ova_max_fail 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.

```
-ova_max_success N
```

Limits the total number of reported successes to *N*. You must supply *N*, otherwise no limit is set. The monitoring of assertions continues, even after the limit is reached.

- ova_simend_max_fail *N*
Terminates the simulation if the number of failures for any assertion reaches *N*. You must supply *N*, otherwise no limit is set.
- ova_success
Enables reporting of successful matches in addition to failures. The default is to report only failures.

The following runtime options enable functional coverage. You can use them only if you compiled with the `-ova_cov` compile-time option.

- ova_cov
Enables functional coverage reporting.
- ova_cov_name *filename*
Specifies the file name or the full path name of the functional coverage report file.
- ova_cov_db *filename*
Specifies the path name of the initial coverage file. VCS needs the initial coverage file to set up the database.

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.

`filter`

Blocks reporting of trivial implication successes. These happen when an implication construct registers a success only because the precondition (antecedent) portion is false (and so the consequent portion is not checked). With this option, reporting only shows successes in which the whole expression matches.

`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 can't 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. This summary looks like this 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's name and location is `./assert.report`, but you can change it by entering the `path/filename` argument. The 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` statements, in addition to failures. The default is to report 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.

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 a CLI Command File

`-i filename`

Specifies a file, containing CLI commands, that VCS executes when simulation starts. After VCS reaches the end of that file, it

takes commands from the standard input. This option is normally used along with the `-s` runtime option and a `+cli+number` compile-time option. A typical file for this option is the `vcs.key` file.

`-k filename | off`

Specifies an alternative name or location for the `vcs.key` file into which VCS writes the CLI and DVE interactive commands that you enter during simulation. The `off` argument tells VCS not to write this file.

`+cliecho`

Specifies that VCS display the CLI commands in a file, which you specify with the `-i` option, as VCS executes these CLI commands.

Options for Specifying VERA Object Files

`+vera_load=filename.vro`

Specifies the VERA object file.

`+vera_mload=filename`

Specifies a text file that contains a list of VERA object files.

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 filename`

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.

`-cm_tglfile filename`

Specifies displaying a total toggle count at runtime for one or more subhierarchies specified by the top-level module instance entered in the file. This option is also a compile-time option.

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. The delayed versions are not available if you use this option at compile-time. See [“Enabling Negative Timing Checks” on page 14-13](#). VCS recognizes `+notimingchecks` to be the same as `+notimingcheck` when you enter it on the `vcs` or `simv` command line.

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} ; see [“Specifying a Very Long Time Before Stopping Simulation” on page 4-8](#).

Options for Recording Output

`-a filename`

Specifies appending all messages from simulation to the bottom of the text in the specified file as well as displaying these messages on the standard 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 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 backannotating delay information from SDF files. This option enables the display of all backannotation warning and error messages.

This default limitation on backannotation 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_annotate` 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_filename*

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

-i *input_filename*

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

-k *key_filename*

Specifies a file where VCS records the UCLI commands it executes. You can use this file as input with the `-i` option in a subsequent 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+filename`

Specifies the name of the output VPD file (default is `vcdplus.vpd`). You must include the full file name with the `.vpd` extension.

`+vpdfilesizesize+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.

`+vpdfileswitchsize+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 +vpdfileswitchsize+10` The first vpd file is named `test.vpd`. When its size reaches 10 MB, 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.

`+vpddrivers`

Reports value changes for all drivers when there is more than one driver. The driver values, for example, enable the Logic Browser to identify which drivers produce an undesired X on the resolved net. This option affects performance and memory usage for larger designs or longer runs.

`+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 Controlling `$gr_waves` System Task

Operations

`-grw filename`

Sets the name of the `$gr_waves` output file to the specified *filename*. The default file name is `grw.dump`.

`+vcs+grwavesoff`

Suppress `$gr_waves` system tasks.

Options for VCD Files

`-vcd filename`

Sets name of `$dumpvars` output file to *filename*. The default file name is `verilog.dump`. A `$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 `$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 Min:Typ:Max 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 `$sdf_annotate` system task; see [“The \\$sdf_annotate System Task” on page 13-3](#). 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. How much more frequently 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 network license if none is available when the job starts.

General Options

Viewing the Compile-Time Options Used to Create the Executable

`-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
simv -E echo > simvE.log
```

You cannot use any other runtime options with the `-E` option.

Stopping Simulation When the Executable Starts

`-s`

Stops simulation just as it begins, and enters interactive mode. Use with the `+cli+number` 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. For an example of checking for a user-defined plusarg runtime option [“Enabling Debugging Features At Runtime” on page 2-10](#).

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. See [“Changing the Timing of a Model” on page 16-16](#).

Specifying `acc_handle_simulated_net` PLI Routine and MIPD Annotation

`+vcs+mipd+noalias`

For the PLI routine `acc_handle_simulated_net`, 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  
| tril | wor | prior | 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
```

``elsif`

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:

```
`elsif 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 . So in the following source code VCS compiles and executes the first block of code and ignores the second block even when you don't 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 Detecting Race Conditions

`\race`

Specifies the beginning of a region in your source code where you want VCS to look for race conditions when you include the `-Xrace=0x1` compile time option. See [“The Dynamic Race Detection Tool” on page 11-2](#).

`\endrace`

Specifies the end of a region in your source code where you want VCS to look for race conditions.

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 backannotation 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_annotate call=sdf_annotate_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 backannotate the delay values for individual bits of a port.

``vcs_mipdnoexpand`

Turns off the enabling of backannotating 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 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, just like 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 Maintaining The File Name 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 Accellera SystemVerilog 3.1 LRM, page 227.

`$error`

Generates a runtime assertion error. See the Accellera SystemVerilog 3.1 LRM, page 227.

`$warning`

Generates a runtime warning message. See the Accellera SystemVerilog 3.1 LRM, page 227.

`$info`

Generates an information message. See the Accellera SystemVerilog 3.1 LRM, page 227.

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 Accellera SystemVerilog 3.1 LRM, page 228.

`$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 Accellera SystemVerilog 3.1 LRM, page 228.

`$asserton`

Tells VCS to resume the monitoring of assertions that it stopped monitoring due to a previous `$assertoff` or `$assertkill` system task. See the Accellera SystemVerilog 3.1 LRM, page 228.

System Tasks for SystemVerilog Assertions

`$onehot`

Returns true if only one bit in the expression is true. See the Accellera SystemVerilog 3.1 LRM, page 228.

`$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 Accellera SystemVerilog 3.1 LRM, page 228.

`$isunknown`

Returns true if one of the bits in the expression has an X value. See the Accellera SystemVerilog 3.1 LRM, page 228.

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 `-I` or `-PP` compile-time options.

`$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 when ever there is an interrupt, such as when VCS executes a `$stop` system task or when you halt VCS using the “.” (period) CLI command, 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;
```

`$vcdplusdeltacycleon`

Enables delta cycle recording in the VPD file for post-processing.

Syntax:

```
$vcdplusevent(net_or_reg, "event_name",  
"<E|W|I><S|T|D>");
```

Displays, in DVE, 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.

`$vcdplusdumpportsoff`

Tells VCS to suspend writing to VPD file the transition times and values of the module instance specified by

`$vcdplusdumpportson` system task. You can use

`$vcdplusdumpportsoff` system task with arguments, but it is not required. **Syntax:**

```
$vcdplusdumpportsoff(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 whole 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's 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 Nth dimension.

dimNRsb

This is an optional argument that specifies the right bound of the Nth 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.

`$vcdplusememorydump`

Records (dumps) a snapshot of the values in a memory or multi-dimensional array into the VPD file. Syntax is the same as the `$vcdplusemenon` system task.

`$vcdpluseoff`

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:

```
$vcdpluseoff[(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).

`$vcdpluseon`

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:

```
$vcdpluseon[(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). Its 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 11364-2001 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 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 [“The \\$recrem Timing Check Syntax” on page 14-11](#) for more information. Also see IEEE Std 1364-2001 pages 246-248.

`$removal`

Reports a timing violation if a 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);
```

See [“The \\$setuphold Timing Check Extended Syntax”](#) on page 14-7 for more information.

`$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$array` to `$syncnorplane`

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. See [Avoiding the Debugging Problems From Port Coercion](#).

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 to 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_annotate`

Tells VCS to backannotate delay values from an SDF file to your Verilog design. See [“The \\$sdf_annotate System Task” on page 13-3](#).

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, also see [“Save and Restart” on page -4-4](#).

`$restart`

Restores the simulation to the state that you saved in the check file with the `$save` system task. Enter this system task at the CLI prompt instead of in the source code. You can also do this by entering the name of the check file at the system prompt. See IEEE std 1364-2001 pages 742-743; also see [“Save and Restart” on page -4-4](#).

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(hierarchical_name, level_number)
```

hierarchical_name

Hierarchical name of the module instance, that is, the top-level instance of the subhierarchy for which you want to enable checking.

level_number

Number of levels down in the subhierarchy from the specified module instance. Checking is also enabled for the instances on these levels.

`$xzcheckoff`

Suppress the warning message every time VCS evaluates a conditional expression to have an X or Z value.

Syntax:

`$xzcheckoff(hierarchical_name, level_number)`

hierarchical_name

Hierarchical name of the module instance, that is, the top-level instance of the subhierarchy for which you want to enable checking.

level_number

Number of levels down in the subhierarchy from the specified module instance. Checking is also enabled for the instances on these levels.

IEEE Standard System Tasks Not Yet Implemented in VCS

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_erlang`

`$dist_t`

`$nochange`

E

PLI Access Routines

VCS comes with a number of access routines. The following access routines are described in this appendix:

- [Access Routines for Reading and Writing to Memories](#)
- [Access Routines for Multidimensional Arrays](#)
- [Access Routines for Probabilistic Distribution](#)
- [Access Routines for Returning a String 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 comes with the 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 E-4 for details.

`acc_getmem_int`

Reads an integer value from specific bits in a Verilog memory word. See "[acc_getmem_int](#)" on page E-5 for details.

`acc_clearmem_int`

Clears a memory, that is, writes zeros to all bits. See "[acc_clearmem_int](#)" on page E-6 for details.

`acc_setmem_hexstr`

Writes a hexadecimal string value to specific bits in a Verilog memory word. See "[acc_setmem_hexstr](#)" on page E-11 for details.

`acc_getmem_hexstr`

Reads a hexadecimal string value from specific bits in a Verilog memory word. See "[acc_getmem_hexstr](#)" on page E-15 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 E-16](#) for details.

`acc_getmem_bitstr`

Reads a bit string from specific bits in a Verilog memory word. See ["acc_getmem_bitstr" on page E-17](#) for details.

`acc_handle_mem_by_fullname`

Returns the handle used by `acc_readmem`. See ["acc_handle_mem_by_fullname" on page E-18](#) for details.

`acc_readmem`

Reads a data file and writes the contents to a memory. See ["acc_readmem" on page E-19](#) for details.

`acc_getmem_range`

Returns the upper and lower limits of a memory. See ["acc_getmem_range" on page E-21](#) for details.

`acc_getmem_size`

Returns the number of elements (or words or addresses) in a memory. See ["acc_getmem_size" on page E-22](#) for details.

`acc_getmem_word_int`

Returns the integer of a memory element. See ["acc_getmem_word_int" on page E-23](#) 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 E-24](#) 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		
	Type	Name	Description
Arguments:	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 left-most 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:	acc_getmem_int acc_setmem_hexstr acc_getmem_hexstr acc_setmem_bitstr acc_getmem_bitstr acc_clearmem_int acc_handle_mem_by_fullname acc_readmem acc_getmem_range acc_getmem_size acc_getmem_word_int acc_getmem_word_range		

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>		
	Type	Description	
Returns:	int	Integer value of the bits in the memory word.	
	Type	Name	Description
Arguments:	handle	memhand	Handle to memory
	int	row	The memory array index
	int	start	Bit number of the left-most 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:	acc_setmem_int acc_setmem_hexstr acc_getmem_hexstr acc_setmem_bitstr acc_getmem_bitstr acc_clearmem_int acc_handle_mem_by_fullname acc_readmem acc_getmem_range acc_getmem_size acc_getmem_word_int acc_getmem_word_range		

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>		
	Type	Description	
Returns:	void		
	Type	Name	Description
Arguments:	handle	memhand	Handle to 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		

Examples

The following code examples illustrate how to use `acc_getmem_int`, `acc_setmem_int`, and `acc_clearmem_int`:

- Example D-1 shows C code that includes a number of functions to be associated with user-defined system tasks.
- Example D-2 shows the PLI table for associating these functions with these system tasks.

- Example D-3 shows the Verilog source code containing these system tasks.

Example D-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 D-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 D-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	
	void		
Arguments:	Type	Name	Description
	handle	memhand	Handle to memory
	char *	hexStrValue	Hexadecimal string
	int	row	The memory array index
	int	start	Bit number of the left-most bit in the memory word where this routine starts writing the string.
Related routines:	acc_setmem_int acc_getmem_int acc_getmem_hexstr acc_setmem_bitstr acc_getmem_bitstr acc_clearmem_int acc_handle_mem_by_fullname acc_readmem acc_getmem_range acc_getmem_size acc_getmem_word_int acc_getmem_word_range		

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 D-4 shows the C source code for an application that calls `acc_setmem_hexstr`.
- Example D-5 shows the contents of a data file read by the application.
- Example D-6 shows the PLI table file that associates the user-defined system task in the Verilog code with the application.
- Example D-7 shows the Verilog source that calls the application.

Example D-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 D-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 D-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 D-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 D-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 D-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		
	Type	Name	Description
Arguments:	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 left-most 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:	acc_setmem_int acc_getmem_int acc_setmem_hexstr acc_setmem_bitstr acc_getmem_bitstr acc_clearmem_int acc_handle_mem_by_fullname acc_readmem acc_getmem_range acc_getmem_size acc_getmem_word_int acc_getmem_word_range		

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 left-most bit in the memory word where this routine starts writing the string.
Related routines:	acc_setmem_int acc_getmem_int acc_setmem_hexstr acc_getmem_hexstr acc_getmem_bitstr acc_clearmem_int acc_handle_mem_by_fullname acc_readmem acc_getmem_range acc_getmem_size acc_getmem_word_int acc_getmem_word_range		

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		
	Type	Name	Description
Arguments:	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 left-most 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:	acc_setmem_int acc_getmem_int acc_setmem_hexstr acc_getmem_hexstr acc_setmem_bitstr acc_clearmem_int acc_handle_mem_by_fullname acc_readmem acc_getmem_range acc_getmem_size acc_getmem_word_int acc_getmem_word_range		

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 <code>acc_readmem</code> only		
Syntax:	<code>acc_handle_mem_by_fullname (fullMemInstName)</code>		
	Type	Description	
Returns:	handle	Handle to the instance	
	Type	Name	Description
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	
	void		
Arguments:	Type	Name	Description
	handle	memhandle	Handle returned by <code>acc_handle_mem_by_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 D-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 D-9 The PLI Table File

```
$test_acc_readmem call=test_acc_readmem
```

Example D-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:	<code>acc_getmem_range (memhandle, p_left_index,p_right_index)</code>		
	Type	Description	
Returns:	void		
	Type	Name	Description
Arguments:	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	Description
Arguments:	handle	memhandle	Handle to 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_word_int acc_getmem_word_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	Description
Arguments:	handle	memhandle	Handle to a memory
	int	row	The element (word address, or row) in the 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_range		

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:	<code>acc_getmem_word_range (memhandle, lsb, len)</code>		
Returns:	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 multidimensional 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 multidimensional array:

`tf_mdanodeinfo` and `tf_imdanodeinfo`

Returns access parameter node information from a multidimensional array. See "[tf_mdanodeinfo and tf_imdanodeinfo](#)" on page E-26 for details.

`acc_get_mda_range`

Returns all the ranges of the multidimensional array. See "[acc_get_mda_range](#)" on page E-28 for details.

`acc_get_mda_word_range`

Returns the range of an element in a multidimensional array. See "[acc_get_mda_word_range\(\)](#)" on page E-29 for details.

`acc_getmda_bitstr`

Reads a bit string, including X and Z values, from an element in a multidimensional array. See "[acc_getmda_bitstr\(\)](#)" on page E-31 for details.

`acc_setmda_bitstr`

Writes a bit string, including X and Z values, from an element in a multidimensional array. See "[acc_setmda_bitstr\(\)](#)" on page E-32 for details.

tf_mdanodeinfo and tf_imdanodeinfo

You use these routines to access parameter node information from a multidimensional array.

tf_mdanodeinfo(), tf_imdanodeinfo()			
Synopsis:	Returns access parameter node information from a multidimensional array.		
Syntax:	<code>tf_mdanodeinfo(nparam, mdanodeinfo_p)</code> <code>tf_imdanodeinfo(nparam, mdanodeinfo_p, instance_p)</code>		
Returns:	Type	Description	
	<code>mdanodeinfo_p *</code>	The value of the second argument if successful; 0 if an error occurs	
Arguments:	Type	Name	Description
	<code>int</code>	<code>nparam</code>	Index number of the multidimensional 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 multidimensional 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 `vcuser.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 multidimensional array.

acc_get_mda_range()			
Synopsis:	Gets all the ranges of the multidimensional array.		
Syntax:	<code>acc_get_mda_range(mdaHandle, size, msb, lsb, dim, plndx, prindex)</code>		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	handle	mdaHandle	Handle to the multidimensional array
	int *	size	Pointer to the size of the multidimensional array
	int *	msb	Pointer to the most significant bit of a range
	int *	lsb	Pointer to the least significant bit of a range
	int *	dim	Pointer to the number of dimensions in the multidimensional array
	int *	plndx	Pointer to the left index of a range
int *	prndx	Pointer to the right index of a range	
Related routines:	tf_mdanodeinfo and tf_imdanodeinfo acc_get_mda_word_range acc_getmda_bitstr acc_setmda_bitstr		

If you have a multidimensional 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 multidimensional array.

acc_get_mda_word_range()			
Synopsis:	Gets the range of an element in a multidimensional array.		
Syntax:	<code>acc_get_mda_range(mdaHandle, msb, lsb)</code>		
	Type	Description	
Returns:	void		
	Type	Name	Description
Arguments:	handle	mdaHandle	Handle to the multidimensional 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:	tf_mdanodeinfo and tf_imdanodeinfo acc_get_mda_range acc_getmda_bitstr acc_setmda_bitstr		

If you have a multidimensional 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 multidimensional array.

acc_getmda_bitstr()			
Synopsis:	Gets a bit string from a multidimensional array.		
Syntax:	<code>acc_getmda_bitstr(mdaHandle, bitStr, dim, start, len)</code>		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	handle	mdaHandle	Handle to the multidimensional array
	char *	bitStr	Pointer to the bit string
	int *	dim	Pointer to the dimension in the multidimensional array
	int *	start	Pointer to the start element in the dimension
int *	len	Pointer to the length of the string	
Related routines:	tf_mdanodeinfo and tf_imdanodeinfo acc_get_mda_range acc_get_mda_word_range acc_setmda_bitstr		

If you have a multidimensional 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 multidimensional array.

acc_setmda_bitstr()			
Synopsis:	Sets a bit string in a multidimensional 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 multidimensional array
	char *	bitStr	Pointer to the bit string
	int *	dim	Pointer to the dimension in the multidimensional 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 multidimensional 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 comes with 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 E-34](#) for details.

`vcs_random_const_seed`

Returns a random number and takes an integer argument. See ["vcs_random_const_seed" on page E-35](#) for details.

`vcs_random_seed`

Returns a random number and takes a pointer to integer argument. See ["vcs_random_seed" on page E-35](#) for details.

`vcs_dist_uniform`

Returns random numbers uniformly distributed between parameters. See ["vcs_dist_uniform" on page E-36](#) for details.

`vcs_dist_normal`

Returns random numbers with a specified mean and standard deviation. See ["vcs_dist_normal" on page E-37](#) for details.

`vcs_dist_exponential`

Returns random numbers where the distribution function is exponential. See ["vcs_dist_exponential" on page E-38](#) for details.

vcs_dist_poisson

Returns random numbers with a specified mean. See ["vcs_random" on page E-34](#) for details.

These routines are declared in the vcs_acc_user.h file in \$VCS_HOME/lib.

vcs_random

You use this routine to obtain a random number.

vcs_random()			
Synopsis:	Returns a random number.		
Syntax:	vcs_random()		
Returns:	Type	Description	
	int	Random number	
Arguments:	Type	Name	Description
	None		
Related routines:	vcs_random_const_seed vcs_random_seed vcs_dist_uniform vcs_dist_normal vcs_dist_exponential vcs_dist_poisson		

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_randon_const_seed			
Synopsis:	Returns a random number.		
Syntax:	<code>vcs_random_const_seed(integer)</code>		
	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:	<code>vcs_random_seed(seed)</code>		
	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>		
	Type	Description	
Returns:	int	Random number	
	Type	Name	Description
Arguments:	int *	seed	Pointer to a seed integer value.
	int	start	Starting parameter for distribution range.
	int	end	Ending parameter for distribution range.
Related routines:	<code>vcs_random vcs_random_const_seed vcs_random_seed vcs_dist_normal vcs_dist_exponential vcs_dist_poisson</code>		

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:	<code>vcs_dist_normal(seed, mean, standard_deviation)</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.
	int	standard_deviation	An integer that is the standard deviation from the mean for the normal distribution.
Related routines:	<code>vcs_random vcs_random_const_seed vcs_random_seed vcs_dist_uniform vcs_dist_exponential vcs_dist_poisson</code>		

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:	<code>vcs_dist_poisson(seed, mean)</code>		
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.
Related routines:	<code>vcs_random vcs_random_const_seed vcs_random_seed vcs_dist_uniform vcs_dist_normal vcs_dist_exponential</code>		

Access Routines for Returning a String 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 E-42](#) for details.

`acc_lsi_dumpports_call`

Monitors instance ports. See "[acc_lsi_dumpports_call](#)" on [page E-43](#) for details.

`acc_lsi_dumpports_close`
Closes specified VCDE files. See "[acc_lsi_dumpports_close](#)" on [page E-45](#) for details.

`acc_lsi_dumpports_flush`
Flushes cached data to the VCDE file on disk. See "[acc_lsi_dumpports_flush](#)" on [page E-46](#) for details.

`acc_lsi_dumpports_limit`
Sets the maximum VCDE file size. See "[acc_lsi_dumpports_limit](#)" on [page E-47](#) for details.

`acc_lsi_dumpports_misc`
Processes miscellaneous events. See "[acc_lsi_dumpports_misc](#)" on [page E-48](#) for details.

`acc_lsi_dumpports_off`
Suspends VCDE file dumping. See "[acc_lsi_dumpports_off](#)" on [page E-49](#) for details.

`acc_lsi_dumpports_on`
Resumes VCDE file dumping. See "[acc_lsi_dumpports_on](#)" on [page E-50](#) for details.

`acc_lsi_dumpports_setformat`
Specifies the format of the VCDE file. See "[acc_lsi_dumpports_setformat](#)" on [page E-52](#) 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 E-53](#) 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 D-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_DUMPPOINTS_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 D-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_DUMPPOINTS_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 dumpports (extended VCD) files. For example, your miscellaneous routine would do the following:

```
my_task_misc(int data, int reason)
{
    acc_lsi_dumpports_misc(data, reason);
    ...
}
```

acc_lsi_dumpports_close

Syntax

```
int acc_lsi_dumpports_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 `$dumpports` and `$lsi_dumpports` or the routine `acc_lsi_dumpports_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 D-13 Example of acc_lsi_dumpports_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_DUMPPOINTS_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 D-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_DUMPPOINTS_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 D-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_DUMPPOINTS_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 D-16 Example of 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 filename is NULL dumping is suspended on all open files.

Returns

The number of files that matched.

Example D-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_DUMPPOINTS_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 D-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_DUMPPOINTS_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_DUMPPOINTS_FORMAT_IEEE
```

```
USE_DUMPPOINTS_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 D-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_DUMPPOINTS_FORMAT_IEEE);
if (acc_lsi_dumpports_call(instance, outfile1)) {
    /* error */
}

/* use LSI format for this file */
```

```
acc_lsi_dumpports_setformat(USE_DUMPPOINTS_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 D-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_dumpports_vhdl_enable(1);
acc_lsi_dumpports_call(instance, outfile1);

/* Exclude VHDL drivers from this report */
acc_lsi_dumpports_vhdl_enable(0);
acc_lsi_dumpports_call(instance, outfile1);

...

```

Caution

This routine has precedence over the `+dumpports+vhdl+enable` and `+dumpports+vhdl+disable` runtime options.

Access Routines for Line Callbacks

VCS comes with 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 E-55](#) 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 E-57](#) for details.

`acc_mod_lcb_enabled`

Tests to see if line callbacks is enabled. See ["acc_mod_lcb_enabled" on page E-58](#) for details.

`acc_mod_lcb_fetch`

Returns an array of breakable lines. See "[acc_mod_lcb_fetch](#)" on [page E-59](#) for details.

`acc_mod_lcb_fetch2`

Returns an array of breakable lines. See "[acc_mod_lcb_fetch2](#)" on [page E-60](#) for details.

`acc_mod_sfi_fetch`

Returns the source file composition for a module. See "[acc_mod_sfi_fetch](#)" on [page E-62](#) 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 D-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 D-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's 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 D-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 D-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 D-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 D-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 E-68](#) 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 E-68](#) 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 true so that subsequent calls to `vcsSpWriteString()` and `vcsSpWriteChar()` will have their data encrypted. See ["vcsSpEncodeOn" on page E-69](#) for details.

`vcsSpEncoding`

This routine gets the current state of encoding. See ["vcsSpEncoding" on page E-71](#) for details.

`vcsSpGetFilePtr`

This routine just returns the value previously passed to the `vcsSpSetFilePtr()` routine. See ["vcsSpGetFilePtr" on page E-72](#) for details.

`vcsSpInitialize`

Allocates a source protect object. See ["vcsSpInitialize" on page E-73](#) for details.

`vcsSpOvaDecodeLine`

Decrypts one line. See ["vcsSpOvaDecodeLine" on page E-74](#) for details.

`vcsSpOvaDisable`

Switches to regular encryption. See ["vcsSpOvaDisable" on page E-75](#) for details.

`vcsSpOvaEnable`

Enables the OpenVera assertions (OVA) encryption algorithm. Tells VCS's encrypter to use the OVA IP algorithm. See ["vcsSpOvaEnable" on page E-76](#) for details.

`vcsSpSetDisplayMsgFlag`

Sets the DisplayMsg flag. See ["vcsSpSetDisplayMsgFlag" on page E-78](#) for details.

`vcsSpSetFilePtr`

Specifies the output file stream. See ["vcsSpSetFilePtr" on page E-78](#) for details.

`vcsSpSetLibLicenseCode`

Sets the OEM license code. See ["vcsSpSetLibLicenseCode" on page E-79](#) for details.

`vcsSpSetPliProtectionFlag`
Sets the PLI protection flag. See ["vcsSpSetPliProtectionFlag" on page E-80](#) for details.

`vcsSpWriteChar`
Writes one character to the protected file. See ["vcsSpWriteChar" on page E-81](#) for details.

`vcsSpWriteString`
Writes a character string to the protected file. See ["vcsSpWriteString" on page E-83](#) for details.

Example D-27 outlines the basic use of the source protection routines.

Example D-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's 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 D-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.
2. 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 D-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 D-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 D-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 D-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 D-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 D-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 D-35 Example of vcsSpOvaDisable

```

#include "enclib.h"
#include "encint.h"

int write_error = 0;
vcsSpStateID esp;

if ((esp = vcsSpInitialize()) 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 D-36 Example of *vcsSpOvaEnable*

```
#include "enclib.h"
#include "encint.h"

int write_error = 0;
vcsSpStateID esp;

if ((esp = vcsSpInitialize()) 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 D-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 D-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 D-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 D-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 D-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 E-69](#)) the specified character is encrypted as it is written to the output file.

If encoding is disabled (see ["vcsSpEncodeOff" on page E-68](#)) 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 E-78](#)) or if there was an error writing to the output file (out-of-disk-space, etc.)

Returns 0 if the write was successful.

Example D-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 E-69](#)) the specified string is encrypted as it is written to the output file.

If encoding is disabled (see ["vcsSpEncodeOff" on page E-68](#)) 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 E-78](#)) or if there was an error writing to the output file (out-of-disk-space, etc.)

Returns 0 if the write was successful.

Example D-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 t is less than the current simulation time, VCS returns control to the calling routine.

Index

Symbols

- a filename 4-13, C-13
- ams_discipline B-56
- ams_iereport B-56
- as assembler B-50
- ASFLAGS B-50
- assert 20-21, 20-23, 23-36, B-31
- B B-62
- C B-50, B-53
- c B-51
- CC B-51
- cc B-51
- CFLAGS B-51
- cm 23-44
- cm_assert_cov 23-47
- cm_assert_cov_cover 23-47
- cm_assert_grade_instances 23-47
- cm_assert_grade_module 23-47
- cm_assert_hier 23-44
- cm_assert_map 23-48
- cm_assert_merge 23-48
- cm_assert_name 23-44, 23-45, 23-47
- cm_assert_report 23-48
- cm_fsmopt allowTemp B-26
- cm_fsmopt optimist B-27
- cm_fsmopt report2StateFsms B-27
- cm_fsmopt reportvalues B-27
- cm_fsmopt reportWait B-27
- cm_fsmopt reportXassign B-27
- cm_fsmresetfilter B-27
- cm_line contassign B-28
- cm_opfile B-28
- cm_scope B-29
- cpp B-52
- debug B-30
- debug_all B-30
- doc B-9
- e 20-45, 23-46
- e name_for_main 17-34, B-41
- E program runtime option C-22
- F filename B-38
- f filename 1-15, B-38
- file B-39
- gen_asm B-50
- gen_c B-51
- gen_obj B-62
- grw C-18
- gui B-30
- h B-9
- help B-9
- I D-19
- i 9-13, C-15
- i filename B-21, C-9
- ID B-49
- jnumber_of_CPUs B-8, B-52

- k 9-13, C-10, C-15
- l C-15
- l filename 1-15, 1-19, 4-13, B-60, C-13
- ld linker B-50
- LDFLAGS B-50
- line 1-15, B-17
- lmc-hm B-61
- lmc-swift 16-16, B-45
- lmc-swift-template 16-4, B-45
- lname B-51
- load 17-33, B-41
- Mdelete B-7
- Mdirectory B-7
- Mfilename B-7
- Minclude B-7
- Mlcmd B-7
- Mloadlist B-8
- Mmakefile B-8
- Mmakeprogram B-8
- Mrelative_path B-8
- Msrclist B-9
- Mupdate B-6
- negdelay B-34, B-36
- noIncComp B-9
- ntb B-11
- ntb_opts B-12
- ntb_sfname B-14
- ntb_vipext B-14
- ntb_vl B-15
- o name B-62
- O number B-53
- O0 B-53
- ova_cov 20-19, 20-24, 23-40, 23-41, B-22, C-6
- ova_cov_db 20-24, 20-46, 23-41, C-6
- ova_cov_events 20-46
- ova_cov_grade_instances 20-46
- ova_cov_grade_modules 20-46
- ova_cov_hier 20-19, 23-40, B-22
- ova_cov_map 20-46
- ova_cov_merge 20-47
- ova_cov_name 20-24, 23-41, C-6
- ova_cov_report 20-47
- ova_debug 20-19, 23-40, B-22
- ova_debug_vpd B-22
- ova_dir 20-20, 23-40
- ova_enable_diag 20-20, 23-40, B-22
- ova_file 20-20, 23-40, B-22
- ova_filter 20-22, 23-40, C-5
- ova_filter_past 20-20, 23-40, B-22
- ova_inline 20-21, B-23
- ova_max_fail 20-22, 23-40, C-5
- ova_max_success 20-22, 23-40, C-5
- ova_name C-4
- ova_quiet 20-21, 23-40, C-4
- ova_report 20-22, 23-40, C-4
- ova_simend_max_fail 20-23, 23-41, C-6
- ova_success 20-23, 23-41
- ova_verbose 20-22, 23-40, C-4
- ovac 20-19, B-22
- override_timescale B-58
- P pli.tab B-41
- parameters 3-13, B-57
- platform B-62
- PP B-30, D-19
- pvalue 3-13, B-56
- q B-46, C-14
- R 1-16, B-21
- s 1-16, B-21, C-22
- sv_pragma 23-42, B-11
- sysc B-58
- syslibs libs B-51
- u B-61
- ucli B-30, C-15
- V B-46, C-14
- v B-4
- vcd filename C-18
- vera 1-17, B-23
- vera_dbind B-23
- Vt B-47
- Xman 25-12, 25-15, B-54
- Xmangle B-54

-Xnoman B-55
 -Xnomangle B-55
 -y 1-17, B-4
 "A" specifier of abstract access 18-7
 "C" specifier of direct access 18-7
 \$ token 23-5
 \$assert_monitor 23-64, D-25
 \$assert_monitor_off 23-64, D-26
 \$assert_monitor_on 23-64, D-26
 \$assertkill D-11
 \$assertoff D-11
 \$asserton D-11
 \$async\$and\$array D-36
 \$bitstoreal D-28
 \$countdrivers D-40
 \$countones 23-68
 \$deposit D-40
 \$disable_warnings D-33
 \$display D-28
 \$display statement 20-75
 \$dist_exponential D-38
 \$dist_normal D-38
 \$dist_poisson D-38
 \$dist_uniform D-38
 \$dumpall D-12
 \$dumpfile D-13
 \$dumpflush D-13
 \$dumplimit D-13
 \$dumpoff D-12
 \$dumpon D-13
 \$dumpports D-15
 \$dumpportsall D-17
 \$dumpportsflush D-17
 \$dumpportslimit D-18
 \$dumpportsoff D-16
 \$dumpportson D-16
 \$dumpvars 3-4, D-13
 \$enable_warnings D-33
 \$error D-11
 \$fatal D-11
 \$fclose D-29
 \$fdisplay D-29
 \$fell 23-10
 \$ferror D-29
 \$fflush D-14, D-29
 \$fflushall D-14
 \$fgetc D-29
 \$fgets D-30
 \$finish D-33
 \$fmonitor D-30
 \$fopen D-30
 \$fread D-30
 \$fscanf D-30
 \$fseek D-30
 \$fstobe D-30
 \$ftell D-30
 \$fwrite D-30
 \$getpattern D-40
 \$gr_waves D-14
 \$hold D-34
 \$info D-11
 \$isunknown 23-68
 \$itor D-28
 \$log D-27
 \$lsi_dumpports 2-34–2-39, D-15
 \$monitor D-28
 \$monitoroff D-29
 \$monitoron D-29
 \$nolog D-28
 \$onehot 23-68
 \$onehot0 23-68
 \$ova_current_time 20-76
 \$ova_global_time_unit 20-76
 \$ova_severity_action 20-76
 \$ova_start 20-78
 \$ova_start levels 20-76
 \$ova_start_time 20-76
 \$ova_stop 20-79
 \$ova_stop levels 20-76
 \$ovadumpoff 20-77
 \$ovadumpon 20-77
 \$past 23-22

\$period D-34
 \$printrtimescale D-32
 \$q_add D-36
 \$q_exam D-37
 \$q_full D-37
 \$q_initialize D-37
 \$q_remove D-37
 \$random 2-33, D-38
 \$readmemb D-31
 \$readmemh D-31
 \$realtime D-37
 \$realtobits D-28
 \$recovery D-34
 \$recrem D-35
 \$removal D-35
 \$reset D-38
 \$reset_count D-39
 \$reset_value D-39
 \$restart D-41
 \$root 22-54
 \$rose 23-9
 \$rtoi D-28
 \$save D-41
 \$sdf_annotate D-39
 \$setup D-35
 \$setuphold D-36
 \$skew D-36
 \$sreadmemb D-31
 \$sreadmemh D-31
 \$stable 23-10
 \$stime D-37
 \$stop D-32
 \$strobe D-29
 \$sync\$nor\$plane D-36
 \$system D-27
 \$systemf D-27
 \$test\$plusargs D-39
 \$time D-37
 \$timeformat D-32
 \$ungetc D-31
 \$value\$plusargs 4-10
 \$vcdplusautoflushoff 6-6
 \$vcdplusautoflushon 6-6
 \$vcdplusdeltacycleoff 6-23
 \$vcdplusdeltacycleon 6-22
 \$vcdplusevent 6-24
 \$vcdplusflush 6-5
 \$vcdplusglitchoff 6-24
 \$vcdplusglitchon 6-23
 \$vcdplusmemoff 6-7
 \$vcdplusmemon 6-7
 \$vcdplusmemorydump 6-7
 \$vcdplusoff 6-4
 \$vcdplustraceoff 6-21
 \$vcdplustraceon 6-21
 \$warning D-11
 \$width D-36
 \$write D-29
 \$writememb D-32
 \$writememh D-32
 %= 22-31
 %CELL 17-15, 17-17
 use of 13-13
 %TASK 17-15
 &= 22-31
 **NC 11-8
 *= 22-31
 +abstract 18-82
 +acc+level_number 17-22, B-19
 +ad B-56
 +allhdrs 18-82
 +allmtm 13-38, B-32, B-35
 +applylearn 17-25–17-29, B-20
 +auto2protect 25-6, B-54
 +auto3protect 25-6, B-54
 +autoprotect 25-6, B-54
 +charge_decay B-32
 +cli+level_number B-17
 +cliecho C-10
 +cliedit B-18
 +csdf+precomp+dir 13-8, B-35
 +csdf+precomp+ext 13-8, B-35

+csdf+precompile 13-7, B-35
 +define+macro=value 1-15, B-61
 +delay_mode_distributed 12-21, B-33
 +delay_mode_path 12-21, B-32
 +delay_mode_unit 12-22, B-32
 +delay_mode_zero 12-22, B-32
 +deleteprotected 25-11, B-54
 +enable_solver_trace 21-40
 +enable_solver_trace_on_failure 21-41
 +error+n 21-30
 +incdir 1-15, B-5
 +libext B-5
 +liborder B-5
 +librescan B-5
 +libverbose B-5, B-45
 +lint B-46
 +list 18-82
 +maxdelays 13-38, 13-39, 16-16, B-33, B-35, C-19
 +memcbk B-60
 +memopt B-58
 +mindelays 13-38, 13-39, 16-16, B-33, B-35, C-19
 +multisource_int_delays 12-6, 13-13, B-33, B-34
 +nbaopt B-33
 +neg_tchk 14-13, 14-20, B-43
 +no_notifier 14-13, B-42, C-12
 +no_pulse_msg C-14
 +no_tchk_msg 14-13, B-43, C-12
 +nocelldefinepli+0 B-48
 +nocelldefinepli+1 B-48
 +nocelldefinepli+2 B-48
 +noerrorIOPCWM B-59
 +nolibcell B-48
 +nospecify 1-15, 14-13, B-42
 +notimingcheck 1-16, 1-20, 4-12, 14-13, 14-14, B-42, C-12
 +ntb_cache_dir 21-37, C-2
 +ntb_debug_on_error 21-38, C-2
 +ntb_enable_solver_trace 21-38, C-3
 +ntb_enable_solver_trace_on_failure C-3
 +ntb_enable_solver_trace_on_failure=value 21-38, C-3
 +ntb_engable_solver_trace C-3
 +ntb_exit_on_error 21-38, 21-39, C-3
 +ntb_load 21-28, C-3
 +ntb_random_seed 21-39, C-3
 +ntb_random_seed=value 21-39
 +ntb_solver_mode C-4
 +ntb_solver_mode=value 21-40, C-4
 +ntb_stop_on_error C-4
 +NTC2 14-19, B-44
 +old_ntc B-43
 +oldsdf 13-5, B-36
 +optconfigfile 3-35, 3-37, B-16
 +overlap 14-23, B-44
 +override_model_delays 16-16, C-23
 +pathpulse B-42
 +pli_unprotected 25-11, B-54
 +plusarg_ignore B-39
 +plusarg_save B-39
 +plus-options C-23
 +prof B-37
 +protect file_suffix B-54
 +pulse_e/number 12-8, 12-9, 12-12, 12-17, 12-18, B-40
 +pulse_int_e 12-7, 12-8, 12-10, 12-12, B-40
 +pulse_int_r 12-7, 12-8, 12-10, 12-12, B-40
 +pulse_on_detect 12-18, B-40
 +pulse_on_event 12-17, B-40
 +pulse_r/number 12-8, 12-9, 12-12, 12-17, 12-18, B-40
 +putprotect+target_dir 25-10, B-54
 +race B-20
 +race=all 11-13, B-21
 +race_maxvecsize 11-5, B-21
 +racecd B-21
 +rad 3-35, B-16
 +sdf_nocheck_celltype 13-16, B-36
 +sdfprotect 25-9
 +sdfprotect file_suffix B-54

+sdfverbose C-14
 +spl_read B-59
 +systemverilogext B-15
 +timopt 13-40
 +transport_int_delays 12-7, 12-10, 12-12, B-34
 +transport_path_delays 12-7, 12-9, 12-12, B-34
 +typdelays 13-38, 13-39, 16-16, B-33, B-36, C-20
 +v2k 1-17, B-58
 +vc 18-81, B-43
 +vcdfile B-31
 +vcs+boundscheck 3-20, B-59
 +vcs+dumppoff+t+ht C-18
 +vcs+dumpon+t+ht C-18
 +vcs+finish 4-8, C-13
 +vcs+flush+all C-21
 +vcs+flush+dump C-21
 +vcs+flush+fopen C-21
 +vcs+flush+log C-21
 +vcs+grwavesoff C-18
 +vcs+ignorestop 4-12, C-23
 +vcs+initmem B-16
 +vcs+initreg B-16
 +vcs+learn+pli 1-20, 17-25–17-29, C-22
 +vcs+lic+vcsi B-49, C-21
 +vcs+lic+wait B-49, C-21
 +vcs+mipd+noalias C-23
 +vcs+mipdexpand B-36
 +vcs+nostdout C-15
 +vcs+saif_libcell 15-2
 +vcs+stop 4-8, C-13
 +vcsi+lic+vcs B-49, C-21
 +vcsi+lic+wait B-49
 +vera_enable_checker_trace_on_failure 21-40
 +vera_load C-10
 +vera_mload C-10
 +vera_solver_model 21-41
 +vera_stop_on_error 21-40, C-4
 +verilog1995ext B-15
 +verilog2001ext B-15
 +vissymbols B-37
 +vpdbufsize 6-26
 +vpddrivers 6-28
 +vpdfile 6-26, B-31
 +vpdfilesize 6-26
 +vpdfileswitchsize B-31
 +vpdignore 6-27
 +vpdnocompress 6-28
 +vpdnostrengths 6-29
 +vpdports 6-28
 +vpi B-41
 +warn 2-34, B-47
 /= 22-31
 <<<= 22-31
 <<= 22-31
 -= 22-31
 =+ 22-31
 >>= 22-31
 >>>= 22-31
 ^= 22-31
 |= 22-31
 ‘celldefine D-2
 ‘default_nettype D-3
 ‘define D-3
 ‘delay_mode_distributed D-6
 ‘delay_mode_path D-5
 ‘delay_mode_unit D-6
 ‘delay_mode_zero D-6
 ‘else D-3
 ‘elseif D-3
 ‘endcelldefine D-2
 ‘endif D-4
 ‘endprotect D-7
 ‘endprotected D-8
 ‘endrace D-5
 ‘ifdef D-4
 ‘ifndef D-5
 ‘include D-8
 ‘line D-10

- 'noportcoerce 2-15, D-8
- 'nounconnected_drive D-10
- 'portcoerce D-8
- 'protect D-8
- 'protected D-8
- 'race D-5
- 'resetall D-3
- 'timescale D-9
- 'unconnected_drive D-10
- 'undef D-5
- 'uselib D-9
- 'vcs_mipdexpand D-7
- 'vcs_mipdnoexpand D-7

A

- a filename 4-13, C-13
- "A" specifier of abstract access 18-7
- +abstract 18-82
- abstract access for C/C++ functions
 - access routines for 18-31–18-77
 - enabling with a compile-time option 18-82
 - using 18-29–18-77
- +acc+level_number 17-22, B-19
- ACC capabilities 17-27
 - cbk 17-13, 17-19
 - cbka 17-14
 - frc 17-14, 17-19
 - gate 17-14
 - mip 17-14
 - mipb 17-14
 - mp 17-14
 - prx 17-14
 - r 17-13, 17-18
 - rw 17-13, 17-18
 - s 17-14
 - specifying 17-11–17-20
 - tchk 17-14
- ACC capabilities for SDF back annotation 13-10–13-13
- access routines for abstract access of C/C++ functions 18-31–18-77
- action blocks 23-28

- action_block 20-75
- +ad B-56
- alias file 8-9
- alias file, default 8-9
- +allhdrs 18-82
- +allmtm 13-38, B-32, B-35
- allvariables 9-6
- always_comb block 22-35
- always_ff block 22-38
- always_latch block 22-38
- ams_discipline B-56
- ams_iereport B-56
- and operator 23-10
- anding sequences 23-10
- annotation
 - overhead 13-13
- aop
 - advice
 - before/after/around 24-143
 - dominates 24-134
 - extends directive 24-130
 - placement element
 - after 24-138
 - around 24-138
- +applylearn 17-25–17-29, B-20
- arb 21-10
- arb.v 21-10
- args PLI Specificaction 17-9
- array
 - output and inout argument type 18-22
- arrays
 - indexing and slicing 22-16
- as assembler B-50
- ASFLAGS options B-50
- assembly code generation
 - assembling by hand B-50
 - passing options to the assembler B-50
 - specifying B-50
 - specifying another assembler B-50
- assert 20-21, 20-23, 23-36, B-10, B-31, C-6
- assert OVA directive 20-3
- assert statements 23-23–23-24

- \$assert_monitor 23-64, D-25
- \$assert_monitor_off 23-64, D-26
- \$assert_monitor_on 23-64, D-26
- assertion classes 21-63
- Assertion failure, displaying message 20-75
- assertion files 20-4
- assertion pragmas 20-60
- assertions 20-4
- assertions, monitoring 20-70
- \$assertkill D-11
- \$assertoff D-11
- \$asserton D-11
- \$async\$and\$array D-36
- +auto2protect 25-6, B-54
- +auto3protect 25-6, B-54
- +autoprotect 25-6, B-54

B

- B B-62
- behavioral drivers 22-20
- benefits of OpenVera Assertions 20-2
- bit
 - C/C++ function argument type 18-10
 - C/C++ function return type 18-9
 - input argument type 18-21
 - output and inout argument type 18-22
 - reg data type in two-state simulation 18-6
- bit data type 22-2
- \$bitstoreal D-28
- break 9-5
 - usage in VSG 24-126
- break and continue
 - break 24-126
- break, CLI command 9-7
- building OVA post-processor 20-26
- byte data type 22-2

C

- C B-50, B-53
- c 21-10, B-51

- C code generating
 - halt before compiling the generated C code B-53
 - passing options to the compiler B-51
 - specifying B-51
 - specifying another compiler B-51
 - suppressing optimization for faster compilation B-53
- C compiler, environment variable specifying the A-3
- "C" specifier of direct access 18-7
- C/C++ functions
 - argument direction 18-8, 18-9
 - argument type 18-8, 18-10
 - calling 18-12–18-14
 - declaring 18-6–18-12
 - extern declaration 18-7
 - in a Verilog environment 18-5–18-6
 - return range 18-8
 - return type 18-7, 18-9
 - using abstract access 18-29–18-77
 - access routines for 18-31–18-77
 - using direct access 18-20–18-29
 - examples 18-22–18-27
- call PLI specification 17-9
- calling C/C++ functions in your Verilog code 18-12–18-14
- call-stack traversal 9-9
- case
 - usage in VSG 24-125
- case statements 24-125
- cbk ACC capability 17-13, 17-19
- cbka ACC capability 17-14
- CC B-51
- cc B-51
- %CELL
 - use of 13-13
- 'celldefine D-2
- CFLAGS B-51
- char data type 22-2
- char*
 - direct access for C/C++ functions
 - formal parameter type 18-20

- char**
 - direct access for C/C++ functions
 - formal parameter type 18-20
- +charge_decay B-32
- check 21-32
- check argument to -ntb_opts B-12
- check PLI specification 17-9
- class
 - methods of 24-63
 - accessing 24-63
 - properties of 24-62
 - accessing 24-63
- CLI 9-13
 - summary 9-14
- +cli+level_number B-17
- CLI commands, OVAPP 20-31
- CLI task invocation 20-76
- cli_0, OVAPP CLI command 20-31
- +cliecho C-10
- +cliedit B-18
- clock signals 13-40–13-44
- clocking block with virtual interfaces 24-211, 24-212
- cm 23-44, B-23, C-9, C-10
- cm_assert_cov 23-47
- cm_assert_cov_cover 23-47
- cm_assert_dir C-9
- cm_assert_grade_instances 23-47
- cm_assert_grade_module 23-47
- cm_assert_hier 23-44, B-24
- cm_assert_map 23-48
- cm_assert_merge 23-48
- cm_assert_name 23-44, 23-45, 23-47
- cm_assert_report 23-48
- cm_cond B-24
- cm_constfile B-26
- cm_dir B-26, C-11
- cm_fsmcfg B-26
- cm_fsmopt B-26
- cm_fsmopt allowTmp B-26
- cm_fsmopt optimist B-27
- cm_fsmopt report2StateFsms B-27
- cm_fsmopt reportvalues B-27
- cm_fsmopt reportWait B-27
- cm_fsmopt reportXassign B-27
- cm_fsmresetfilter B-27
- cm_glitch C-11
- cm_hier B-27
- cm_ignorepragmas B-28
- cm_libs B-28
- cm_line contassign B-28
- cm_log C-11
- cm_name B-28, C-12
- cm_noconst B-28
- cm_opfile B-28
- cm_pp B-28
- cm_scope B-29
- cm_tgl B-30, B-60
- cm_tglfile B-29, C-12
- cnt.txp 20-5
- command alias file 8-9
- command line interface 9-13
- commands, list of 8-4
- compiler directives D-2–D-10
- compiler options
 - f
 - syntax 21-30
- compile-time options B-1–B-62
 - +error 21-30
 - ntb 21-29
 - ntb_cmp 21-26, 21-30
 - ntb_define macro 21-30
 - ntb_noshell 21-27
 - ntb_opts 21-32
 - ntb_sfname 21-26
 - ntb_shell_only 21-27
 - ntb_sname 21-26
 - ntb_spath 21-27, 21-31
 - ntb_vipext 21-31
 - ntb_vl 21-28, 21-32
- compiling
 - incremental compilation 3-3–3-7
 - shared 3-5–3-7

- triggering 3-4
- verbose messages B-46
- compiling, design, design, testbench and Verilog module 21-27
- compiling, design, shell, and Verilog module 21-27
- concurrent assertions 23-1
- consecutive repetition 23-6
- context-dependent pragmas 20-60
- continue
 - syntax 24-127
- control constructs 21-4
- control tasks, OVA debug 20-77
- \$countdrivers D-40
- cover OVA directive 20-3
- cover statements 23-25–23-28
- coverag_group
 - embedded
 - syntax for defining 21-47
 - syntax for defining 21-47
- Coverage 20-42
- coverage
 - closed-loop analysis 21-46
 - coverage_load() 21-58, 24-243
 - single coverage_group 21-58, 24-243
 - cumulative 24-226
 - get_coverage() 21-51
 - get_inst_coverage() 21-51
 - instance-based 24-227
 - loading coverage data
 - coverage_instance() 21-59, 24-244
 - loading embedded coverage data
 - coverage_instance() 21-59, 24-244
 - open-loop analysis 21-46
- coverage expressions 20-3
- coverage, testing 20-6
- coverage_group 21-46
 - predefined functions 24-230
 - inst_query() 24-230, 24-232, 24-233
- coverage_group_attributes
 - at_least 21-49
- coverage_instance() 21-59, 24-244
- coverage_load 21-58, 24-243

- cpp B-52
- +csdf+precomp+dir 13-8, B-35
- +csdf+precomp+ext 13-8, B-35
- +csdf+precompile 13-7, B-35

D

- daidir directory 3-7
- .daidir extension 3-7
- data PLI specification 17-9
- Data Type Mapping File
 - VCS/SystemC cosimulation interface 19-27
- debug B-30
- Debug Control Tasks, OVA 20-77
- debug_all B-30
- debugging
 - capability 9-13
- debugging with CLI 9-13
- debugging, OVA 20-37
- declaring C/C++ functions in your Verilog code 18-6–18-12
- default alias file 8-9
- 'default_nettype D-3
- 'define D-3
- +define+macro=value 1-15, B-61
- delay values
 - back annotating to your design D-39
- +delay_mode_distributed 12-21, B-33
- 'delay_mode_distributed D-6
- +delay_mode_path 12-21, B-32
- 'delay_mode_path D-5
- +delay_mode_unit 12-22, B-32
- 'delay_mode_unit D-6
- +delay_mode_zero 12-22, B-32
- 'delay_mode_zero D-6
- delete, CLI command 9-8
- +deleteprotected 25-11, B-54
- Delta Cycle Information
 - Capturing delta cycle information 6-22
- dep_check 21-33, 21-36
- dep_check argument to -ntb_opts B-12

- \$deposit D-40
- design, compiling 21-27
- DEVICE SDF construct 13-28
- direct access for C/C++ functions
 - examples 18-22–18-27
 - formal parameters
 - types 18-20
 - rules for parameter types 18-20–18-22
 - using 18-20–18-81
- Direct Access Interface directory 3-7
- direction of a C/C++ function argument 18-9
- directive, assert 20-3
- directive, cover 20-3
- directory
 - .daidir 3-7
- \$disable_warnings D-33
- Disk space
 - temporary 3-24
- \$display statement 20-75
- \$display D-28
- DISPLAY_VCS_HOME A-3
- \$dist_exponential D-38
- \$dist_normal D-38
- \$dist_poisson D-38
- \$dist_uniform D-38
- DKI communication 19-6
- do while statement 22-34
- doc B-9
- double*
 - direct access for C/C++ functions
 - formal parameter type 18-20
- downstack, CLI command 9-10
- dump files 20-35
- \$dumpall D-12
- \$dumpfile D-13
- \$dumpflush D-13
- dumping signals automatically 20-37
- \$dumplimit D-13
- \$dumpoff D-12
- \$dumpon D-13
- \$dumpports D-15
- \$dumpportsall D-17

- \$dumpportsflush D-17
- \$dumpportslimit D-18
- \$dumpportsoff D-16
- \$dumpportson D-16
- \$dumpvars 3-4, D-13
- DVE
 - starting 5-7

E

- e 20-45, 23-46
- e name_for_main 17-34, B-41
- E program C-22
- 'else D-3
- 'elseif D-3
- \$enable_warnings D-33
- enabling
 - only where used in the last simulation 17-27
- 'endcelldefine D-2
- ended method 23-13
- 'endif D-4
- 'endprotect D-7
- 'endprotected D-8
- 'endrace D-5
- enum construct 22-5
- enumerations 22-5
- Environment variables 1-8–1-9, A-2–A-4
- environment variables
 - VCS_CC 18-85
 - VCS_CPP 18-85
 - VCS_LD 18-85
- \$error D-11
- error messages, OVA linter option 20-9
- error messages, OVA MR linter option 20-16
- event 9-7
- Event Access 9-11
- Event coverage, testing 20-6
- event expression/structure with virtual interfaces 24-213
- example
 - temporal assertion file 20-4, 20-5

- extends directive
 - advice 24-131
 - introduction 24-131
- extern declaration 18-7
- extern declarations 18-27

F

- f
 - syntax 21-30
- F filename B-38
- f filename 1-15, B-38
- facilities, test 20-2
- \$fatal D-11
- \$fclose D-29
- \$fdisplay D-29
- \$ferror D-29
- \$fflush D-14, D-29
- \$fflushall D-14
- \$fgetc D-29
- \$fgets D-30
- file B-39
- file, report 20-41
- Files
 - tokens.v B-55
 - vcs.key 9-13
- files, temporal assertion 20-4
- \$finish D-33
- flow 20-7
- flow control 21-3
- flow of OVAPP 20-25
- \$fmonitor D-30
- \$fopen D-30
- four state Verilog data
 - stored in vec32 18-15
- frc ACC capability 17-14, 17-19
- \$fread D-30
- \$fscanf D-30
- \$fseek D-30
- \$fstobe D-30
- \$ftell D-30

- full_case 20-59, 20-60
- Functional Coverage 20-42
- functions
 - void 22-44
- functions and system tasks, OVA 20-68
- functions in SystemVerilog 22-41
- \$fwrite D-30

G

- gate ACC capability 17-14
- gen_asm B-50
- gen_c B-51
- gen_obj B-62
- get_coverage() 21-51
- \$getpattern D-40
- gmake 1-9, A-2
- \$gr_waves D-14
- grw C-18
- gui B-30

H

- h B-9
- hard 24-92
- help B-9
- help 9-2
- \$hold D-34

I

- I D-19
- i 9-13, C-15
- i filename B-21, C-9
- ID B-49
- 'ifdef D-4
- if-else
 - production definition 24-123
 - usage in VSG 24-123
- 'ifndef D-5
- ignore 22-70, B-10

- Ignoring Calls and License Checking 6-27, C-17
- implications 23-19
- implicit .* connections 22-58
- implicit .name connections 22-58
- +incdir 1-15, B-5
- 'include D-8
- Incremental Compilation 3-5–3-7, B-6–B-9
- \$info D-11
- info, CLI command 9-2
- ignore 22-70, B-10
- inlining
 - context-dependent pragmas 20-60
- inout
 - C/C++ function argument direction 18-9
- inout ports 22-20
- input
 - C/C++ function argument direction 18-9
- input ports
 - valid data types 22-56
- inst_query() 24-230, 24-232, 24-233
- int
 - C/C++ function argument type 18-10
 - C/C++ function return type 18-9
 - direct access for C/C++ functions
 - formal parameter type 18-20
 - input argument type 18-21
 - output and inout argument type 18-22
- int data type 22-2
- int*
 - direct access for C/C++ functions
 - formal parameter type 18-20
- Interactive Debugging
 - example 9-11
- interface, command line 9-13
- Interfaces 22-62–22-69
- interfaces
 - functions in 22-68
 - methods 22-68
 - modports 22-66
- intrinsic timing delay backannotation 13-28
- introducing OpenVera Assertions 20-2

- invoking DVE 5-7
- IOPATH entries in SDF files 13-13
- \$itor D-28

J

- jnumber_of_CPUs B-8, B-52

K

- k 9-13, C-10, C-15
- keywords
 - after 24-138
 - around 24-138
 - before 24-138
 - extends 24-130
 - hard 24-92
 - hide 24-158, 24-160
 - local 24-62
 - new 24-41
 - program 21-7
 - protected 24-62
 - randseq 24-118
 - static 24-45
 - this 24-64
 - virtuals 24-157

L

- l C-15
- l filename 1-15, 1-19, 4-13, B-60, C-13
- ld linker B-50
- LDFLAGS options B-50
- +libext B-5
- +liborder B-5
- +librescan B-5
- +libverbose B-5, B-45
- line 1-15, B-17
- 'line D-10
- line, CLI command 9-3
- linking B-51
 - linking a specified library to the executable B-51

- linking by hand B-51
 - passing options to the linker B-50
 - specifying another linker B-50
- +lint B-46
- Lint, using 3-10–3-12
- linter rules, OVA code checking 20-9
- linter rules, OVA MR code checking 20-16
- +list 18-82
- list, CLI command 9-3
- lmc-hm B-61
- lmc-swift 16-16, B-45
- lmc-swift-template 16-4, B-45
- lname B-51
- load 17-33, B-41
- loading the testbench 21-28
- loads 9-5
- local 24-62, 24-63
- \$log D-27
- log file, environment variable specifying the A-4
- log files
 - specifying compilation log file 1-15, B-60
 - specifying with a system task D-27
- logic data type 22-2
- longint data type 22-2
- \$lsi_dumpports 2-34–2-39, D-15

M

- mailboxes, CLI command 9-7
- Main Window
 - example 5-3
- make 1-9, A-2
- make program B-8
- Marchive B-6
- maxargs PLI specification 17-10
- +maxdelays 13-38, 13-39, 16-16, B-33, B-35, C-19
- MDAs, system tasks and functions 6-7
- Mdelete B-7
- Mdir 3-5

- Mdirectory B-7
- +memcbk B-60
- +memopt B-58
- memories
 - sparse memory models 2-25
- memory size limits 2-25
- Menu Bar
 - using 5-7
- message, assertion failure 20-75
- method
 - local 24-63
 - public 24-63
- methods 22-68
- Mfilename B-7
- minargs PLI specification 17-9
- Minclude B-7
- +mindelays 13-38, 13-39, 16-16, B-33, B-35, C-19
- mip ACC capability 17-14
- mipb ACC capability 17-14
- misc PLI specification 17-9
- Mldcmd B-7
- Mlib 3-5, B-7, B-9
- Mloadlist B-8
- Mmakefile B-8
- Mmakeprogram B-8
- modports 22-66
- modports in virtual interfaces 24-208, 24-209
- module path delays
 - disabling for an instance 13-39
 - suppressing 1-15, B-42
 - in specific module instances 13-40
- \$monitor D-28
- Monitoring assertions 20-70
- \$monitoroff D-29
- \$monitoron D-29
- mp ACC capability 17-14
- Mrelative_path B-8
- Msrclist B-9
- multi-dimensional arrays 22-15
- multiple OVAPP post-processing sessions 20-32

+multisource_int_delays 12-6, 13-13, B-33, B-34
-Mupdate B-6

N

native code generating
 specifying B-62
Native Testbench
 in VCS 9-13
+nbaopt B-33
-nbt_v1 21-32
**NC 11-8
+neg_tchk 14-13, 14-20, B-43
-negdelay B-34, B-36
new() 24-62
 user-defined, syntax 24-43
next, CLI command 9-3
no_case 20-60
no_file_by_file_pp 21-10, 21-33
no_file_by_file_pp argument to -ntb_opts B-12
+no_identifier C-12
+no_notifier 14-13, B-42
+no_pulse_msg C-14
+no_tchk_msg 14-13, B-43, C-12
+nocelldefinepli+1 B-48
nocelldefinepli PLI specification 17-10
+nocelldefinepli+0 B-48
+nocelldefinepli+2 B-48
-noIncrComp B-9
+nolibcell B-48
\$nolog D-28
'noportcoerce 2-15, D-8
+nospecify 1-15, 14-13, B-42
-notice B-46
+notimingcheck 1-16, 1-20, 4-12, 14-13, 14-14, B-42, C-12
'nounconnected_drive D-10
-ntb 21-29, B-11
ntb 21-16
+ntb_cache_dir C-2
-ntb_cmp 21-26, 21-30, B-11
+ntb_debug_on_error C-2
-ntb_define B-11
ntb_define 21-16
-ntb_define macro 21-30
+ntb_enable_solver_trace_on_failure C-3
+ntb_engable_solver_trace C-3
+ntb_exit_on_error C-3
-ntb_filext 21-31, B-12
-ntb_incdir 21-31, B-12
ntb_incdir 21-16
+ntb_load C-3
-ntb_noshell 21-27, B-12
-ntb_opts 21-32, B-12
 check 21-32
 dep_check 21-33, 21-36
 no_file_by_file_pp 21-33
 print_deps 21-34, B-13
 rvm 21-34
 tb_timescale 21-34
 use_sigprop 21-34
 vera_portname 21-35
-ntb_opts no_file_by_file_pp 21-44
-ntb_opts rvm 21-98
ntb_opts rvm 21-98
+ntb_random_seed C-3
-ntb_sfname 21-26, B-14
-ntb_shell_only 21-27, B-14
-ntb_sname 21-26, B-14
+ntb_solver_mode C-4
-ntb_spath 21-27, 21-31, B-14
+ntb_stop_on_error C-4
-ntb_vipext 21-31, 21-45, B-14
-ntb_vl 21-28, B-15
+NTC2 14-19, B-44
null comparison in virtual interfaces 24-213

O

-o name B-62
-O number B-53
-O0 B-53

- object data members 9-9
- +old_ntc B-43
- +oldsdf 13-5, B-36
- OpenVera Assertions
 - benefits 20-2
 - flow 20-7
 - introduction 20-2
 - overview 20-4
- operating system commands, executing D-27
- OpenVera assertion classes 21-63
- +optconfigfile 3-35, 3-37, B-16
- or operator 23-11
- oring sequences 23-11
- output
 - C/C++ function argument direction 18-9
- output ports
 - valid data types 22-56
- OVA cover directive 20-3
- OVA debug control tasks 20-77
- OVA linter 20-8
- OVA post-processing 20-24
- OVA post-processor
 - building 20-26
 - running 20-26
- OVA system tasks and functions 20-68
- OVA user action function 20-82
- OVA, assert directive 20-3
- OVA, see OpenVera Assertions
- OVA, Verilog parameters in 20-63
- ova_cov 20-19, 20-24, 23-40, 23-41, B-22, C-6
- ova_cov_cover 20-46
- ova_cov_db 20-24, 20-46, 23-41, C-6
- ova_cov_events 20-19, 20-46, 23-40, B-22
- ova_cov_grade_instances 20-46
- ova_cov_grade_modules 20-46
- ova_cov_hier 20-19, 23-40, B-22
- ova_cov_map 20-46
- ova_cov_merge 20-47
- ova_cov_name 20-24, 23-41, C-6
- ova_cov_report 20-47
- \$ova_current_time 20-76
- ova_debug 20-19, 23-40, B-22
- ova_debug_vpd B-22
- ova_dir 20-20, 23-40
- ova_enable_diag 20-20, 23-40, B-22
- ova_file 20-20, 23-40, B-22
- ova_filter 20-22, 23-40, C-5
- ova_filter_past 20-20, 23-40, B-22
- \$ova_global_time_unit 20-76
- ova_inline 20-21, B-23
- ova_lint 20-8, B-23
- ova_lint_magellan 20-16, B-23
- ova_max_fail 20-22, 23-40, C-5
- ova_max_success 20-22, 23-40, C-5
- ova_name C-4
- ova_quiet 20-21, 23-40, C-4
- ova_report 20-22, 23-40, C-4
- \$ova_severity_action 20-76
- ova_simend_max_fail 20-23, 23-41, C-6
- \$ova_start 20-78
- \$ova_start levels 20-76
- \$ova_start_time 20-76
- \$ova_stop 20-79
- \$ova_stop levels 20-76
- ova_success 20-23, 23-41, C-6
- ova_trace_off assertion_hierarchical_name, OVAPP CLI command 20-31
- ova_trace_off instance_hierarchical_name assertion_name time, OVAPP CLI command 20-31
- ova_trace_on assertion_hierarchical_name, OVAPP CLI command 20-32
- ova_trace_on instance_hierarchical_name assertion_name time, OVAPP CLI command 20-31
- ova_verbose 20-22, 23-40, C-4
- ovac 20-19, B-22
- \$ovadumpoff 20-77
- \$ovadumpon 20-77
- OVAPP CLI commands 20-31
- OVAPP Flow 20-25
- +overlap 14-23, B-44
- +override_model_delays 16-16, C-23

-override_timescale B-58

P

-P pli.tab 17-20, B-41

packed arrays 22-14

packed keyword 22-11

packed structure 22-11

parallel compilation B-8, B-52

parallel_case 20-60

parameter expansion, Verilog with OVA 20-64

-parameters 3-13, B-57

path 23-47

+pathpulse B-42

PATHPULSE\$ specparam, enabling B-42

\$period D-34

persistent PLI specification 17-10

placement element

after 24-138

around 24-138

-platform B-62

PLI specifications

args 17-9

call 17-9

check 17-9

data 17-9

maxargs 17-10

minargs 17-9

misc 17-9

nocelldefinepli 17-10

persistent 17-10

size 17-9

PLI table file 17-6–17-21, B-37

+pli_unprotected 25-11, B-54

pli.tab file 17-6–17-21

+plusarg_ignore B-39

+plusarg_save B-39

plusargs, checking for on the simv command line D-39

+plus-options C-23

pointer

C/C++ function argument type 18-10

C/C++ function return type 18-9

input argument type 18-21

output and inout argument type 18-22

port coercion 2-14

Port Mapping File

VCS/SystemC cosimulation interface 19-26

'portcoerce D-8

post_randomize() 24-100

post-processing OVA 20-24

post-processing sessions, multiple OVAPP 20-32

-PP B-30, D-19

pp_fastforward time, OVAPP CLI command 20-31

pp_rewind time, OVAPP CLI command 20-31

pragmas 20-60

for SystemVerilog assertions 23-42

pre_randomize() 24-100

print CLI command 9-3

print this, CLI command 9-9

print variable, CLI command 9-9

print_deps 21-34

print_deps argument to -ntb_opts B-13

printing values 9-9

\$printtimescale D-32

priority case statements 22-34

priority if statements 22-32

procedure_prototype

example 24-155, 24-156

production definitions 24-119

production items 24-119

weights 24-119

production weights 24-122

+prof B-37

properties 23-17–23-22

formal arguments 23-18

implications 23-19

inverting 23-21

past value 23-22

+protect file_suffix B-54

'protect D-8

'protected D-8

prx ACC capability 17-14
public 24-63
+pulse_e/number 12-8, 12-9, 12-12, 12-17,
12-18, B-40
+pulse_int_e 12-7, 12-8, 12-10, 12-12, B-40
+pulse_int_r 12-7, 12-8, 12-10, 12-12, B-40
+pulse_on_detect 12-18, B-40
+pulse_on_event 12-17, B-40
+pulse_r/number 12-8, 12-9, 12-12, 12-17,
12-18, B-40
pulses
 filtering out narrow pulses B-40
 and flag as error B-40
+putprotect+target_dir 25-10, B-54
-pvalue 3-13, B-56

Q

-q B-46, C-14
\$q_add D-36
\$q_exam D-37
\$q_full D-37
\$q_initialize D-37
\$q_remove D-37

R

-R 1-16, B-21
r ACC capability 17-13, 17-18
+race B-20
'race D-5
race conditions
 avoiding 2-2–2-7
 continuous assignment evaluations 2-5
 in counting events 2-6
 in flip-flops 2-4
 setting a value twice at the same time 2-3
 time zero 2-7
 using and setting a value at the same time
 2-2
+race=all 11-13, B-21
+race_maxvecsize 11-5, B-21

+racecd B-21
+rad 3-35, B-16
\$random 2-33, D-38
random() 24-126
randomize() 24-100
randomize() with 24-108
randseq 24-118, 24-127
 syntax to define block 24-118
\$readmemb D-31
\$readmemh D-31
real
 C/C++ function argument type 18-10
 input argument type 18-21
 output and inout argument type 18-22
\$realtime D-37
\$realtobits D-28
recommendation messages, OVA linter option
20-9
recommendation messages, OVA MR linter
option 20-16
\$recovery D-34
\$recrem D-35
Reference Verification Methodology, using
21-98
reg
 C/C++ function argument type 18-10
 C/C++ function return type 18-9
 input argument type 18-21
 output and inout argument type 18-22
repeat
 usage in VSG 24-126
repeat loops 24-126
 random() 24-126
repetition
 consecutive 23-6
report file 20-41
\$reset D-38
\$reset_count D-39
\$reset_value D-39
'resetall D-3
resetting
 keeping track of the number of resets D-39

- passing a value from before to after a reset D-39
- resetting VCS to simulation time 0 D-38
- \$restart D-41
- results 20-41, 20-42
- return range of a C/C++ function 18-8
- return type of a C/C++ function 18-7, 18-9
- reverse()
 - example 24-165, 24-166, 24-168, 24-169, 24-170, 24-171, 24-172, 24-173, 24-174, 24-175, 24-176
 - syntax 24-165, 24-166, 24-167, 24-168, 24-169, 24-170, 24-171, 24-172, 24-173, 24-174, 24-175, 24-176
- \$rtoi D-28
- rules, OVA linter option 20-9, 20-16
- running OVA post-processor 20-26
- runtime libraries, environment variable specifying the A-4
- runtime option
 - +ntb_load 21-28
- runtime options
 - +ntb_cache_dir 21-37
 - +ntb_debug_on_error 21-38
 - +ntb_enable_solver_trace_on_failure 21-38
 - +ntb_engable_solver_trace 21-38
 - +ntb_exit_on_error 21-39
 - +ntb_load 21-39
 - +ntb_random_seed 21-39
 - +ntb_solver_mode 21-40
 - +ntb_stop_on_error 21-40
 - enable_solver_trace 21-40
 - enable_solver_trace_on_failure 21-41
 - vera_enable_checker_trace 21-40
 - vera_enable_checker_trace_on_failure 21-40
 - vera_solver_mode 21-41
- rvm 21-34, 21-98
- rw ACC capability 17-13, 17-18

S

- s 1-16, B-21, C-22
- s ACC capability 17-14

- \$save D-41
- scalar
 - direct access for C/C++ functions
 - formal parameter type 18-20
- scalar*
 - direct access for C/C++ functions
 - formal parameter type 18-20
- scopes 9-5
- \$sdf_annotate D-39
- +sdf_nocheck_celltype 13-16, B-36
- SDFPOLICY setup variable 13-32
- +sdfprotect 25-9
- +sdfprotect file_suffix B-54
- +sdfverbose C-14
- semaphores, CLI command 9-7
- sequences 23-3–23-14
 - anding 23-10
 - conditions for 23-12
 - end point 23-13
 - formal arguments 23-5
 - oring 23-11
 - repetition 23-6
 - specifying a clock 23-9
 - specifying a range of clock ticks 23-5
 - unconditionally extending 23-6
- sequential devices
 - inferring 2-18–2-21, 13-40–13-44
- set variable, CLI command 9-9
- \$setup D-35
- \$setuphold D-36
- shell, compiling, Verilog module, compiling 21-27
- shortint data type 22-2
- show break, CLI command 9-8
- show thread, CLI command 9-10
- show, CLI command 9-4, 9-6
- signals, dumping automatically 20-37
- simulation state
 - saving D-41
- simulation time
 - setting
 - example 5-9
- simulator graphical user interface 4-2

- simv executable file 1-13
- size PLI specification 17-9
- \$skew D-36
- slicing arrays 22-16
- SmartQ
 - reverse() 24-165, 24-166
- solve-before
 - hard 24-92
- Source Pane
 - Toolbar icon 5-5, 5-6
- sparse memory models 2-25
- specify blocks
 - disabling for an instance 13-39
 - suppressing 1-15, B-42
 - in specific module instances 13-40
- specifying libraries on the link line B-51
- \$sreadmemb D-31
- \$sreadmemh D-31
- stack, CLI command 9-9
- starting DVE 5-7
- static 24-45
- step, CLI command 9-3
- \$stimen D-37
- \$stop D-32
- stream generation
 - production definitions 24-118
- stream generator
 - randseq 24-118
- string
 - C/C++ function argument type 18-10
 - C/C++ function return type 18-9
 - input argument type 18-21
 - output and inout argument type 18-22
- \$strobe D-29
- struct construct 22-10
- structural drivers 22-20
- structures 22-10
- suspend_thread 21-4
- sv_pragma 23-42, B-11
- sverilog 22-69, B-9
- SWIFT SmartModels
 - generating a template B-45

- \$sync\$nor\$plane D-36
- sysc 19-14, B-58
- syscan utility 19-8–19-10
- syslib libs B-51
- \$system D-27
- system tasks D-10–D-42
 - IEEE standard system tasks not implemented D-42
- System tasks and functions, OVA 20-68
- SystemC
 - cosimulating with Verilog 19-1
- \$systemf D-27
- SystemVerilog assertions 23-1–23-35
- SystemVerilog functions 22-41
- SystemVerilog tasks 22-40
- +systemverilogext B-15

T

- t 21-10
- task invocation from CLI 20-76
- tasks in SystemVerilog 22-40
- tb_timescale 21-34
- tb_timescale argument to -ntb_opts B-13
- tbreak CLI command 9-8
- tchk ACC capability 17-14
- temporal assertion files 20-4
- temporal assertions 20-4
- terminate, CLI command 9-10
- test facilities 20-2
- \$test\$plusargs D-39
- testbench, compiling 21-27
- testbench, loading 21-28
- testing event coverage 20-6
- this 24-64
- thread, CLI command 9-7, 9-10
- Threads 9-10
- throughout operator 23-12
- \$time D-37
- time
 - simulation

- setting 5-9
- \$timeformat D-32
- timescale B-57
- 'timescale D-9
- timing check system tasks
 - disabling
 - in specific module instances 13-40
- timing check system tasks, disabling 1-16, B-42
- timing checks
 - disabling for an instance 13-39
- Timopt
 - the timing optimizer 13-40–13-44
- +timopt 13-40
- TMPDIR A-3
- tokens.v file 23-71, B-55
- Toolbar
 - using 5-7
- +transport_int_delays 12-7, 12-10, 12-12, B-34
- +transport_path_delays 12-7, 12-9, 12-12, B-34
- trigger, CLI command 9-11
- +typdelays 13-38, 13-39, 16-16, B-33, B-36, C-20
- typedef construct 22-5, 22-11

U

U

- direct access for C/C++ functions
 - formal parameter type 18-20
- u B-61
- U*
 - direct access for C/C++ functions
 - formal parameter type 18-20
- UB*
 - direct access for C/C++ functions
 - formal parameter type 18-20
- UCLI 8-1
- ucli B-30, C-15
- UCLI commands, list 8-4

- unaccelerated
 - definitions and declarations 2-16–2-17
 - structural instance declarations 2-17
- 'unconnected_drive D-10
- 'undef D-5
- \$ungetc D-31
- unions 22-10
- unique case statements 22-33
- Unique Events
 - \$vcdplusevent 6-24
 - \$vcdplusglitchoff 6-24
 - \$vcdplusglitchon 6-23
- unique if statements 22-32
- uniquifying identifier codes in VCD files 7-3
- unpacked arrays 22-14
- unpacked structure 22-11
- up, CLI command 9-4
- upper case characters, changing all identifiers to B-61
- upstack, CLI command 9-10
- use_sigprop 21-34, B-13
- use_sigprop argument to -ntb_opts B-13
- use_vpiobj 17-32, B-41
- 'uselib D-9
- User Action Function, OVA 20-82
- user defined data types 22-5
- utility, vcsplit 7-26

V

- V B-46, C-14
- v B-4
- +v2k 1-17, B-58
- \$value\$plusargs 4-10
- variables 9-5
 - writing to 22-19
- +vc 18-81, B-43
- vc_2stVectorRef() 18-51
- vc_4stVectorRef() 18-50
- vc_argInfo() 18-74
- vc_arraySize() 18-40

- vc_FillWithScalar() 18-72
- vc_get2stMemoryVector() 18-67
- vc_get2stVector() 18-55
- vc_get4stMemoryVector() 18-64
- vc_get4stVector() 18-54
- vc_getInteger() 18-49
- vc_getMemoryInteger() 18-62
- vc_getMemoryScalar() 18-61
- vc_getPointer() 18-47
- vc_getReal() 18-45
- vc_getScalar() 18-40
- vc_handle
 - definition 18-29
 - using 18-30–18-31
- vc_hdrs.h file 18-27–18-28
- vc_Index() 18-75
- vc_Index2() 18-76
- vc_Index3() 18-76
- vc_is2state() 18-36
- vc_is2stVector() 18-38
- vc_is4state() 18-35
- vc_is4stVector() 18-37
- vc_isMemory() 18-34
- vc_isScalar() 18-32
- vc_isVector() 18-33, 18-77
- vc_mdaSize() 18-76
- vc_MemoryElemRef) 18-58
- vc_MemoryRef() 18-56
- vc_MemoryString() 18-69
- vc_MemoryStringF() 18-70
- vc_put2stMemoryVector() 18-67
- vc_put2stVector() 18-55
- vc_put4stMemoryVector() 18-66
- vc_put4stVector() 18-54
- vc_putInteger() 18-49
- vc_putMemoryInteger() 18-64
- vc_putMemoryScalar() 18-62
- vc_putMemoryValue() 18-68
- vc_putMemoryValueF() 18-68
- vc_putPointer() 18-47
- vc_putReal() 18-44
- vc_putScalar() 18-40
- vc_putValue() 18-45
- vc_putValueF() 18-46
- vc_StringToVector() 18-48
- vc_toChar() 18-40
- vc_toInteger() 18-41
- vc_toString() 18-42
- vc_toStringF() 18-43
- vc_VectorToString() 18-49
- vc_width() 18-39
- vcat utility 7-12
- vcd filename C-18
- VCD+ 6-2
 - Advantages 6-2
 - Capturing data 6-3
 - Command line options
 - Buffer size 6-26
 - Bypass data compression 6-28
 - Control maximum file size 6-26
 - Do not store strength information 6-29
 - Ignore \$vcdplus calls in code 6-27
 - Set output file name 6-26
 - Store driver information 6-28
 - Store port information 6-28
 - Managing simulation 6-29
 - System Tasks
 - \$vcdplusautoflushoff 6-6
 - \$vcdplusautoflushon 6-6
 - \$vcdplusdeltacycleoff 6-23
 - \$vcdplusdeltacycleon 6-22
 - \$vcdplusevent 6-24
 - \$vcdplusflush 6-5
 - \$vcdplusglitchoff 6-24
 - \$vcdplusglitchon 6-23
 - \$vcdplusmemoff 6-7
 - \$vcdplusmemon 6-7
 - \$vcdplusmemorydump 6-7
 - \$vcdplusoff 6-4
 - \$vcdpluson
- vcd2vpd B-31
- +vcdfile B-31
- vcdiff utility 7-5
 - syntax 7-6

- vcdpost utility 7-2
 - syntax 7-4
- VCS
 - predefined text macro D-4
 - using Native Testbench 9-13
- vcs 21-10
- vcs command line 1-13
- +vcs+boundscheck 3-20, B-59
- +vcs+dumppoff+t+ht C-18
- +vcs+dumpon+t+ht C-18
- +vcs+finish 4-8, C-13
- +vcs+flush+all C-21
- +vcs+flush+dump C-21
- +vcs+flush+fopen C-21
- +vcs+flush+log C-21
- +vcs+grwavesoff C-18
- +vcs+ignorestop 4-12, C-23
- +vcs+initmem B-16
- +vcs+initreg B-16
- +vcs+learn+pli 1-20, 17-25–17-29, C-22
- +vcs+lic+vcsi C-21
- +vcs+lic+wait B-49, C-21
- +vcs+mipd+noalias C-23
- +vcs+mipdexpand B-36
- +vcs+nostdout C-15
- +vcs+saif_libcell 15-2
- +vcs+stop 4-8, C-13
- VCS/SystemC cosimulation interface
 - compiling for using 19-14, 19-20
 - supported port data types 19-8
- VCS_CC A-3
- VCS_CC environment variable 18-85
- VCS_COM A-3
- VCS_CPP environment variable 18-85
- VCS_LD environment variable 18-85
- VCS_LIC_EXPIRE_WARNING A-4
- VCS_LOG A-4
- `vcs_mipdexpand D-7
- `vcs_mipdnoexpand D-7
- VCS_NO_RT_STACK_TRACE A-4
- VCS_RUNTIME A-4
- VCS_SWIFT_NOTES A-4
- +vcsi+lic+vcs C-21
- +vcsi+lic+wait B-49
- vcs.key file 9-13
- vcsplit utility 7-26
- vec32
 - storing four state Verilog data 18-15
- vec32*
 - direct access for C/C++ functions
 - formal parameter type 18-20
- vera 1-17, B-23
- Vera runtime options
 - vera_solver_mode 21-61
- vera_dbind B-23
- vera_defines.vrh 21-7
- +vera_load C-10
- +vera_mload C-10
- vera_portname 21-35
- vera_portname argument to -ntb_opts B-13
- verify 20-41, 20-42
- Verilog
 - System Tasks
 - \$vcdplustraceoff 6-21
 - \$vcdplustraceon 6-21
- Verilog module, compiling 21-27
- Verilog parameter expansion 20-64
- Verilog parameters in OVA 20-63
- +verilog1995ext B-15
- +verilog2001ext B-15
- violation windows
 - using multiple non-overlapping 14-23–14-27
- Virtual Interface Modports 24-208, 24-209
- Virtual Interfaces 24-202
- +vissymbols B-37
- vlogan utility 19-17–19-19
- VMC 1-22
- void
 - C/C++ function return type 18-9
- void functions 22-44
- void()
 - example 24-88
 - semantics 24-89

- syntax 24-88
- void*
 - direct access for C/C++ functions
 - formal parameter type 18-20
- void**
 - direct access for C/C++ functions
 - formal parameter type 18-20
- VPD
 - Command line options
 - Ignore \$vcdplus calls in code C-17
- VPD files D-18
- vpd2vcd B-31
- +vpdfile B-31
- +vpdfileswitchsize B-31
- +vpi B-41
- VSG
 - if-else usage 24-123
 - overview 24-118
 - production definitions 24-119
 - production weights 24-122
 - randseq 24-118
 - usage of case 24-125
 - usage of repeat 24-126
 - use of break 24-126
- Vt B-47

W

- wait_child 21-4

- wait_var 21-4
- +warn 2-34, B-47
- \$warning D-11
- warning messages, OVA linter option 20-9
- warning messages, OVA MR linter option 20-16
- waveform dump files 20-35
- weights 24-122
- \$width D-36
- wn ACC capability 17-13
- wrapper for VCS/SystemC cosimulation
 - instantiating in SystemC 19-19
 - SystemC in Verilog 19-8
 - Verilog in SystemC 19-17
- \$write D-29
- \$writememb D-32
- \$writememh D-32

X

- Xman 25-12, 25-15, B-54
- Xmangle B-54
- Xnoman B-55
- Xnomangle B-55

Y

- y 1-17, B-4